Tom Miller
Dean Johnson

Foreword by Shawn Hargreaves,
Principal Software Design Engineer,
XNA Game Studio, Microsoft

XNA Game Studio 4.0 Programming

Developing for Windows® Phone 7 and Xbox 360®

Developer's Library

XNA Game Studio 4.0 Programming

Developing for Windows® Phone 7 and Xbox 360®

Developer's Library Series

Stephen G. Kochan
Second Edition
Programming in Objective-C 2.0
A complete introduction to the Objective-C language for Mac OS X and iPhone development
Developer's Library
Shane Conder
Lauren Darcey
Android Wireless Application Development
Developer's Library
Jason Ouellette
Development with the Force.com Platform
Building Business Applications in the Cloud
Developer's Library
Erica Sadun
Second Edition
The iPhone Developer's Cookbook
Building Applications with the iPhone 3.0 SDK
Developer's Library
Addison-Wesley

Visit **developers-library.com** for a complete list of available products

The **Developer's Library Series** from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that's useful for other programmers.

Developer's Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari Books Online

XNA Game Studio 4.0 Programming

Developing for Windows® Phone 7 and Xbox 360®

Tom Miller
Dean Johnson

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data is on file

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978-0-672-33345-3
ISBN-10: 0-672-33345-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana
First printing, December 2010

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Deadline Driven Publishing

Indexer
Erika Millen

Proofreader
Jennifer Gallant

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Composition
Nonie Ratcliff

Contents at a Glance

Introduction **1**

1 Getting Started **5**

2 Sprites and 2D Graphics **13**

3 The Game Object and the Default Game Loop **29**

4 Introduction to 3D Graphics **41**

5 Lights, Camera, Action! **85**

6 Built-In Shader Effects **105**

7 States, Blending, and Textures **141**

8 Introduction to Custom Effects **171**

9 Using the Content Pipeline **215**

10 Having Fun with Avatars **239**

11 Understanding Performance **287**

12 Adding Interactivity with User Input **311**

13 Turn Up the Volume **353**

14 Storage **375**

15 Gamer Services **391**

16 Multiplayer Networking **409**

17 Using Media in XNA Game Studio **441**

A Reach vs. HiDef Chart **455**

B Using the Windows Phone FMRadio **459**

C Windows Phone 7 Launchers and Choosers **463**

D Dealing with Tombstoning **479**

Index **487**

Contents

Foreword xiv
Acknowledgments xv
About the Authors xvi

Introduction 1
So You Want to be a Game Developer? 1
A Brief History of XNA Game Studio 1
What Is Available in Game Studio 4.0? 3
Why This Book? 4

1 Getting Started 5
Installing XNA Game Studio 4.0 5
  Downloading the Tools 6
  App Hub Membership 6
  XNA Game Studio Connect 9
Writing Your First Game 11
  Your First XNA Game Studio Windows Game 11
  Your First XNA Game Studio XNA Xbox 360 Game 11
  Your First XNA Game Studio Windows Phone 7 Game 12
Download Samples 12
Summary 12

2 Sprites and 2D Graphics 13
What Does 2D Mean? 13
Show Me Something on Screen 14
Spritebatch 16
  Drawing 16
  Moving Things Around 19
  Animation 20
  Controlling State 21
Rendering Text 25
Summary 27

3 The Game Object and the Default Game Loop 29
What Is in a New Project? 29
The Game Class 32
    Virtual Methods 32
    Methods 33
    Properties 34
    GameTime 34
Game Loop 36
    Update and Draw 36
Components 38
    GameComponents 38
Summary 40

4 Introduction to 3D Graphics 41
3D Graphics in XNA Game Studio 41
What Are 3D Graphics? 42
Makeup of a 3D Image 42
3D Math Basics 43
    Coordinate Systems 44
    Vectors in 3D Graphics 46
    Matrix 53
Graphics Pipeline 61
    Graphics Card 62
    Vertex Shader 62
    Backface Culling 63
    Viewport Clipping 63
    Rasterization 64
    Pixel Shader 64
    Pixel Tests 64
    Blending 65
    Final Output 65
Reach and HiDef Graphics Profiles 65
    Graphics Profiles Define Platform Capabilities 66
    The Reach Profile 66
    The HiDef Profile 66

Let the 3D Rendering Start 67
  GraphicsAdapter 67
  GraphicsDevice 69
  Drawing with Primitives 71
Summary 83

5 Lights, Camera, Action! 85
Why Do I See What I See? 85
View Matrix 87
Projection Matrix 88
  Perspective 89
  Orthographic 93
Camera Types 93
  Static Cameras 94
Models 95
  What Is a Model? 95
  Rendering Models 99
Summary 103

6 Built-In Shader Effects 105
Using BasicEffect 106
  Basic Lighting 108
  Textures, Vertex Colors, and Fog 114
Using the Effect Interfaces 121
Using DualTextureEffect 122
Using AlphaTestEffect 124
Using EnvironmentMapEffect 124
Using SkinnedEffect 127
Summary 140

7 States, Blending, and Textures 141
Device States 141
  BlendState 142
  DepthStencilState 149
Render Targets 155
  Faking a Shadow with a Depth Buffer and Render Targets 158

Back to Device States 161
    The Stencil Buffer 161
    RasterizerState 164
SamplerStates 166
    Other Texture Types 169
Summary 170

8 Introduction to Custom Effects 171
What Is a Custom Effect? 171
High Level Shading Language 172
Creating Your First Custom Effect 172
Parts of an Effect File 173
    Global Variables 174
    Vertex Structures 174
Drawing with a Custom Effect 177
Vertex Color 179
Texturing 180
    Setting Sampler States in Effect File 183
    Textures Repeating 184
Lighting 186
    Ambient Lighting 186
    Triangle Normals 190
    Diffuse Lighting 192
    Emissive Lighting 198
    Specular Lighting 199
    Fog 202
    Point Lights 206
Effect States 209
    Alpha Blending Using Effect States 211
Summary 213

9 Using the Content Pipeline 215
Tracing Content Through the Build System 215
Content Processors 216
Content Importers 223
Combining It All and Building Assets 226
    Combining What You Learned So Far 235
Summary 238

10 Having Fun with Avatars 239
- Introduction to Avatars 239
  - Accessing Avatar Information Using AvatarDescription 240
  - Loading Avatar Animations with AvatarAnimation 243
  - Drawing the Avatar Using AvatarRenderer 246
- Modifying Avatar Lighting 248
- Playing Multiple Animations 249
- Blending Between Animations 253
- Interacting with Objects 260
- 2D Avatars Using Render Targets 263
- Custom Avatar Animations 265
  - Creating the Custom Animation 266
  - Building the Custom Animation Type 267
  - Creating the Content Processor 273
  - Adding the Custom Animation to Your Game 283
  - Updating Your Game to Use the Custom Animation 284
- Summary 285

11 Understanding Performance 287
- General Performance 287
  - Who Takes Out the Garbage? 289
  - Multithreading 292
- Graphics Performance 293
- Measuring Performance 295
  - Performance Measurement Tools 306
- Cost of Built-In Shaders 307
- Summary 309

12 Adding Interactivity with User Input 311
- Using Input in XNA Game Studio 311
- Polling versus Event-Based Input 312
- The Many Keys Of A Keyboard 312
  - Reading Keyboard State 313
  - Moving Sprite Based on Keyboard Input 315
  - Onscreen Keyboard 316

Precision Control of a Mouse 320
    Reading Mouse State 320
    Moving Sprite Based on Mouse Input 322
    Setting the Mouse Position 324
Xbox 360 Gamepad 324
    Reading Gamepad State 325
    Moving Sprites Based on Gamepad Input 329
    Thumb Stick Dead Zones 332
    Other Types of Controllers 332
    Is the Gamepad Connected? 333
Multitouch Input For Windows Phones 334
    Reading the TouchPanel Device State 334
    Determine Number of Touch Points 336
    TouchPanel Width, Height, and Orientation 337
    Moving Sprite Based on Multitouch Input 337
    Reading Gestures from the TouchPanel 339
    Displaying GestureSample Data 341
Windows Phone Sensors and Feedback 342
    Acceleration Data using the Accelerometer 344
    Locating a Windows Phone with the Location Service 348
    Providing User Feedback using Vibration 351
Summary 351

13 Turn Up the Volume 353
Playing Sound Effects 353
    Using SoundEffect for Audio Playback 354
    Microsoft Cross-Platform Audio Creations Tool (XACT) 360
Dynamic Sound Effects 368
    Recording Audio with a Microphone 368
    Generating Dynamic Sound Effects 371
Summary 374

14 Storage 375
What Is Storage? 375
Isolated Storage 375
    Saving and Loading Data 377
    The IsolatedStorageFile Object 379

XNA Game Studio Storage 380
    Recreating the Project on Xbox 380
    Devices and Containers 382
    Getting a Device 383
    Looking at the API 387
Loading Loose Files from Your Project 388
Summary 390

15 Gamer Services 391
GamerServicesComponent 391
Guide Class 392
    Trial Mode 392
    Now the Bad News 397
    Platform-Specific Guide Functionality 397
Gamers and Profiles 402
    GameDefaults 405
    Presence 406
    Privileges 406
    With Friends Like This... 407
Summary 408

16 Multiplayer Networking 409
Multiplayer Games 409
    Getting Ready for Networking Development 410
    Main Menu and State Management 412
    Creating a Network Session 416
    Building a Game Lobby 423
    Playing the Game 425
    Searching for an Available Network Session 430
    Joining an Available Network Session 435
    Sending Player Invites 438
    Simulating Real World Network Conditions 439
Summary 440

17 Using Media in XNA Game Studio 441
What Is Media? 441
    Playing a Song 441
    MediaPlayer 442
    Songs and Metadata 443
Media Enumeration 444
    Media Library 444
Video 448
    Rendering a Video 448
Visualizations 451
    Rendering Visualization Data 451
Summary 453

A Reach vs. HiDef Chart 455

B Using the Windows Phone FMRadio 459

C Windows Phone 7 Launchers and Choosers 463

D Dealing with Tombstoning 479

Index 487

Foreword

I got my first computer in 1989, when I was 13. It was an Oric-1 with a 1-MHz CPU and 48k RAM. It didn't come with any games, but when you switched it on, up came a screen that said:

Ready

■

It was ready to be programmed, and the manual dived straight into teaching me how to do this:

> First the bad news. ORIC doesn't understand English. But now the good news. You don't have to learn a complicated electronic language, because ORIC speaks a language called BASIC. If your machine is switched on, we'll see how easy this is. Type:
>
> ```
> PRINT "HELLO"
> ```
>
> and then press the [RETURN] key.

Wow! I just made my first program, and the computer did exactly what I told it to do. What a thrill! I was hooked.

A few years later, we upgraded to an Atari ST. This was better than the Oric in all ways but one: bigger, faster, higher resolution. When I switched it on, excited to start programming, I saw a desktop waiting for me to launch an application. Where was the programming language? I was horrified to learn I could not program this machine without first locating and buying an expensive third-party interpreter or compiler. If I had not already learned to program on the Oric, this hurdle would have been too steep, so I would never have bothered to program the Atari, never gotten a job in the games industry, never joined the XNA team, and would not be writing this foreword today.

Learning to program is important for many reasons. As a society, we need skilled programmers to create and maintain the programs that make the modern world work. As a democracy, we need people who understand computers well enough to make sure we control these programs, and not the other way around. And as individuals, programming can be great fun.

I worry that as computers have become more powerful, they also became more intimidating. I think the best thing about XNA Game Studio is how it restores the immediacy and fun I experienced with my Oric. To lower the barriers to entry, we have a free and easy-to-learn programming language called C#. To provide that magical thrill of making the machine do your bidding, we have powerful yet simple APIs, and the ability to run your creations not just on PC, but also Xbox 360 and Windows Phone. And last but not least, to learn how to put all these pieces together, we have books like this one. Nice work Dean and Tom!

I hope you have as much fun making games with XNA as I did with my Oric.

—Shawn Hargreaves, Principal Software Design Engineer,
XNA Game Studio, Microsoft

Acknowledgments

To my wife Tanya, my son Lucas, and my daughter Samantha, who always put up with me doing regardless of what wild things I decide to do. The patience they display allows me to do crazy things like write another book, and I certainly don't let them know how much I appreciate it often enough.

The entire XNA Game Studio team deserves congratulations for getting this release out. I know the hard work that went into it. A few members of the team went above and beyond for this book as well; in particular, Dean (who wrote half of the book), Shawn (who wrote the foreword), and Jason Kepner (who did all of the figures that weren't screenshots).

I'd also like to thank Tony Cox who gave me the chance to fulfill my dream of being a game developer and hired me for my new role in Microsoft Game Studios.

In my first two books, I thanked my mother Vicki, so I suppose I have to continue that tradition as well.

Last, but certainly not least, I'd like to thank you, the reader. I hope you enjoy reading the book as much as I enjoyed writing it.

—Tom Miller

This book is dedicated to the love of my life my wife, Jessica, and to our new baby daughter, Evie-Lyn. Without your love and commitment I would not be the person I am today.

I would like to thank my coauthor Tom. Without Tom, this book would not have been possible.

I would like to thank the entire XNA Game Studio team. Everyone worked tremendously hard on XNA Game Studio 4.0. Specifically, I would like to Shawn Hargreaves for writing the foreword to this book and Jason Kepner for drawing all of the figures for our chapters.

Finally, I would like to thank my mother Donna and stepfather Michael for buying me my first computer and giving me the freedom to explore and break it.

—Dean Johnson

About the Authors

Tom Miller has been with Microsoft for a full decade. He specializes in bringing together managed code and gaming. He wrote and supported Managed DirectX, and for the past few years, he has been largely responsible for implementing the framework (graphics, audio, input, storage, and other core features) included in XNA Game Studio products. He currently works for Microsoft Game Studios.

Dean Johnson joined Microsoft in 2006 and helped launch the XNA Creators Club pipeline allowing hobbyists and independent developers to release their games on the Xbox LIVE Indie Games Marketplace. He currently is a Lead Software Development Engineer working on the XNA Game Studio product team.

Both authors actively blog and participate in game development conferences.

Introduction

So You Want to be a Game Developer?

We've worked in what you would call the "game industry" for years, and during our time, we've met many people and developers. They almost universally share a similar trait in that they either have been or at some time wanted to be a game developer. Games are everywhere—in movies, television, and the Internet. The audience of people playing games has expanded, too, with the popularity of online social media sites such as Facebook.

Much like a book or a movie, games can be a journey. They can tell a story and put the person experiencing it into a whole new world. Unlike a book or a movie, though, games are interactive. The player actually has an effect on the world. People can get immersed in these worlds, and once they are, a natural extension of that is the desire to create other (even better) worlds for others to get immersed in. With this desire to create other worlds, we have a budding game developer.

You may find this surprising, but writing a game is hard work, or, writing a good game is hard work. First, you need to have a good idea, and hopefully, that idea is fun. After you have the idea, though (even if it is the best idea ever), you still have to write the actual core game play and mechanics. After that, there is a ton of work to get something that looks polished. Finishing developing a game is probably one of the most difficult things for someone to do, particularly if he or she isn't aware of how much work is required.

It isn't easy to become a professional game developer. For established publishers and studios, it is difficult to come in off the street and get a job writing games. Publishers and studios want people with experience who have shipped games. Times are changing, however, with platforms that allow self-publishing (such as XNA Game Studio on Xbox or for Windows Phone 7). Anyone can publish a game and actually charge for it to make money.

A Brief History of XNA Game Studio

This book covers XNA Game Studio 4.0, and it has been quite a journey to get to this fourth release. XNA Game Studio 4.0 naturally builds on previous versions of XNA Game Studio, which build on a combination of technologies that go way back.

The technologies go all the way back to Windows 95 as a matter of fact! When Windows 95 was released, Microsoft released something called the Windows Game SDK, which would later be renamed to something you might be more familiar with—DirectX. It is somewhat humorous that the X that is everywhere started off as a joke. The original APIs released in DirectX 1.0 were DirectDraw, DirectInput, and DirectSound. The X was used as shorthand to replace each of the actual API component names to talk about the whole group, and that X soon became the official name. It migrated all the way through to the original Xbox to XNA.

Before DirectX, making games was much more difficult than it is today. There wasn't a standard way of talking to the various pieces of hardware that existed on all the computers, so if you wanted to write a game that worked for everything, you had to write special code for each piece of hardware you wanted to support. With DirectX, there was a standard way of accessing the hardware, and game developers and hardware manufacturers all over rejoiced!

DirectX has gone through quite a few versions, adding new functionality as it developed (such as 3D graphics, networking, and music) and is now on version 11 that shipped with Windows 7. When people talk about DirectX 11, though, they are almost always talking about Direct3D 11, as no other components have changed since DirectX9.

I got ahead of myself, though. Let's backtrack a little to DirectX 7.0. This was the first version of DirectX that included functionality for a language other than C, as it included DirectX for Visual Basic. This was actually when I joined the DirectX team, specifically to work on that portion of the product. I continued to work on it through DirectX 8.0.

DirectX 8.0 was the first version to include programmable shaders, something you read more about in Chapter 8. It's actually hard to believe how far we've come since then, as there isn't any way to write graphics code without shaders! DirectX 8.0 is also the time I began looking at this funny thing called .NET.

DirectX 9.0 was the first release of DirectX that included a component specifically designed for the Common Language Runtime (CLR). This component is Managed DirectX. A lot of work went into that project and although it looked only vaguely familiar to people using DirectX, it fit right in for people using C# and the other managed languages.

The response Managed DirectX received was surprising and a bit overwhelming. Although DirectX for Visual Basic had expanded the development audience, Managed DirectX did so even more. The API was cleaner, easier to use, and felt like all of the other managed components that were out there. The biggest worry then (and one you still hear about today) was related to performance. No one could believe that a managed API (particularly one with a garbage collector) could run fast.

After spending a few years working on Managed DirectX, I left the DirectX team in January of 2006 to join a new group that wanted to develop this thing called XNA, which was eventually released late in 2006 as XNA Game Studio Express.

Game Studio changed all the rules. It took the ease of use and power that Managed DirectX had, made it even easier and more powerful, and added the capability to run games on an Xbox 360. Historically, game consoles have always been closed systems, including the Xbox 360. Before Game Studio, the only way to run code on an Xbox 360 was to be a registered native developer, which required a contract with Microsoft and an approved game!

Much like DirectX, Game Studio kept evolving. With 2.0, support for networking via Xbox LIVE was added. Any version of Visual Studio 2005 was allowed to be used (rather than the C# Express that was required in the first version). At the launch of 3.0, new features arrived with the inclusion of support for Zune and the capability to publish and sell games on Xbox LIVE via the Xbox LIVE Community Games (now called Xbox LIVE Indie Games). Version 3.1 included support for the Zune HD, Video, Xbox LIVE Parties, and Avatars.

Game Studio 4.0 is the latest version where the biggest feature is the addition of the Windows Phone 7 development. There are, of course, other updates, too, including a much simpler graphics API and features such as microphone support and dynamic audio. This version is what this book covers. It has been a wild ride getting here, and I can't wait to see where we go next.

What Is Available in Game Studio 4.0?

Game Studio 4.0 has everything you need to make great and compelling games for Windows Phone 7, Xbox 360, and Windows. The Game Studio 4.0 release is broken into two different profiles: One is called Reach, which encompasses features that exist on all platforms, and the other is called HiDef, which includes extra features that exist only on Xbox 360 and Windows (depending on the graphics card). Each of these areas is discussed in depth later in the book. Table 1 shows the major namespaces and feature areas contained in the framework.

Table 1 **Namespaces included in XNA Game Studio 4.0**

Namespace	Features
Microsoft.Xna.Framework	General framework features, math, and game objects
Microsoft.Xna.Framework.Graphics	All graphics features, including 2D and 3D
Microsoft.Xna.Framework.Audio	All audio features
Microsoft.Xna.Framework.Input	All input features, including game pads, keyboard, and mouse
Microsoft.Xna.Framework.GamerServices	Functionality for accessing portions of the Xbox LIVE services
Microsoft.Xna.Framework.Media	Media features for pictures, music, and so on
Microsoft.Xna.Framework.Content	Content pipeline features
Microsoft.Xna.Framework.Net	All synchronous networking features
Microsoft.Xna.Framework.Storage	Storage features for HiDef

Why This Book?

This book is not only the best reference source for Game Studio 4.0, it also has the added benefit of being authored by two of the people responsible for bringing this product to you. It would be difficult to find two people more knowledgeable on this subject than us, and we want to transfer that knowledge to you.

We cover every aspect of game development using Game Studio 4.0 for every platform that it supports. This includes specifics for APIs that exist on Windows Phone 7 (such as accelerometer and other sensors) that are not part of the Game Studio API, but are important for games.

1

Getting Started

> **Note**
>
> Before we begin teaching you all about XNA Game Studio 4.0 you will first need to get your development environment setup and ready for development. There are slightly different steps depending of if you are targeting the Xbox 360, Windows Phone 7 device, or Windows but we will cover them all.

In this chapter you will learn:

- How to install XNA Game Studio 4.0
- How to sign up for an App Hub Membership
- Windows Phone Marketplace registration
- How to write your first XNA Game Studio games for Windows, Xbox 360, and Windows Phone 7

XNA Game Studio enables you to quickly build games. Before you can start to create your first XNA Game Studio game, you need to install the latest version of the product XNA Game Studio 4.0 and set up your Xbox 360 and Windows Phone 7 device.

Installing XNA Game Studio 4.0

XNA Game Studio 4.0 includes a number of components. The XNA Framework consists of the developer APIs that you use in your game to write code against. Visual Studio project templates and tools are provided for the different XNA project types, including games and game libraries for each of the supported platforms. The content pipeline is used to build game content for use in your game. XNA Game Studio 4.0 also installs a number of tools that you can use throughout this book.

In past releases, XNA Game Studio had its own installer that could install different versions of Visual Studio. XNA Game Studio 4.0 is integrated as part of the Microsoft Windows Phone Developer Tools. Even the Windows and Xbox 360 projects come in the *Microsoft Windows Phone Developer Tools* along with other projects for creating Silverlight-based Windows Phone applications.

The tools install a special version of Visual Studio called Microsoft Visual Studio 2010 Express for Windows Phone. If you have another version of Visual Studio 2010 installed, XNA Game Studio 4.0 is installed into that version, too.

Note

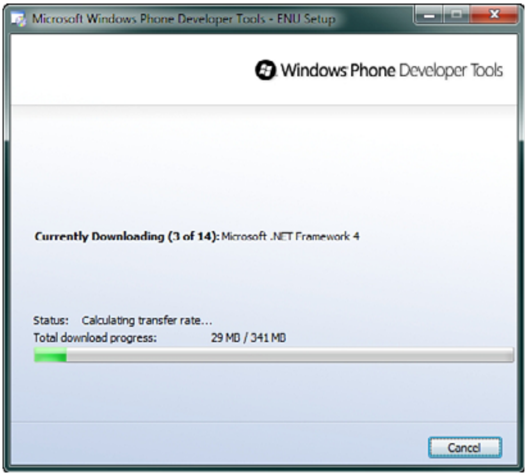
A standalone XNA Game Studio 4.0 installer that can install the Windows and Xbox 360 projects is available at http://go.microsoft.com/fwlink/?LinkId=197288.

Downloading the Tools

The first step to install XNA Game Studio 4.0 is to download the installer from the following link:

http://go.microsoft.com/?linkid=9713250

After downloading the installer, double-click it and follow the instructions to complete the installation. Figure 1.1 shows the Windows Phone Developer Tools installer.



Microsoft Windows Phone Developer Tools - ENU Setup

Windows Phone Developer Tools

Currently Downloading (3 of 14): Microsoft .NET Framework 4

Status: Calculating transfer rate...

Total download progress: 29 MB / 341 MB

Figure 1.1 Windows Phone Developer Tools installer

You might have to restart your PC to complete the initialization.

After the initialization has completed you should have three new top-level menus in your start menu: Microsoft Visual Studio 2010 Express, Microsoft XNA Game Studio 4.0, and Windows Phone Developer Tools.

Clicking Microsoft Visual Studio 2010 Express and then clicking the Microsoft Visual Studio 2010 Express for Windows Phone launches Visual Studio.

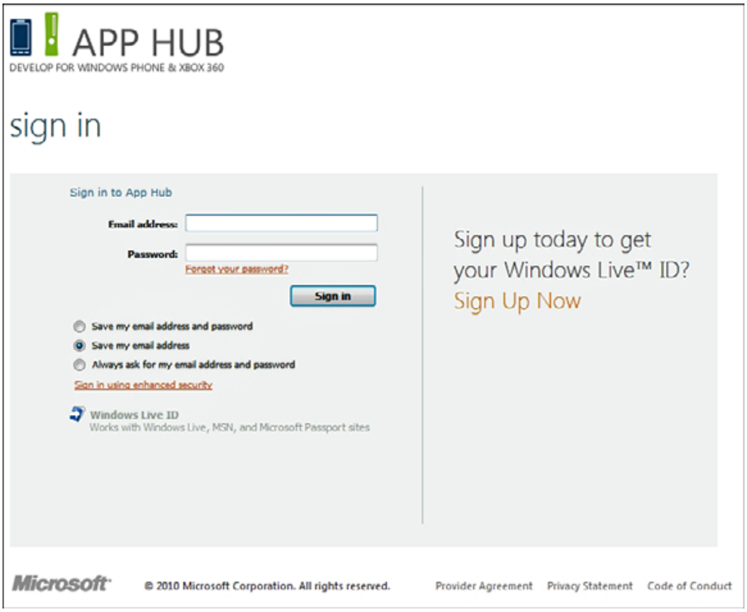
App Hub Membership

To develop XNA games for your Xbox 360 and to deploy games to your Windows Phone 7 device, you need to have an App Hub membership. The membership also allows you as a developer to sell your XNA Xbox 360 games on the Xbox LIVE Indie Games marketplace and your XNA or Silverlight games in the Windows Phone 7 marketplace.

To register on the App Hub website and purchase the App Hub membership first go to the following URL.

https://windowsphone.create.msdn.com/Register/

Sign in with an existing Windows Live ID or sign up for a new one (Figure 1.2).


APP HUB
DEVELOP FOR WINDOWS PHONE & XBOX 360
sign in
Sign in to App Hub
Email address:
Password:
Forgot your password?
Sign in
Windows Live ID
Sign Up Now
Provider Agreement
Privacy Statement
Code of Conduct

○ Save my email address and password
● Save my email address
○ Always ask for my email address and password

Sign in using enhanced security

Works with Windows Live, MSN, and Microsoft Passport sites

Sign up today to get your Windows Live™ ID?

Microsoft © 2010 Microsoft Corporation. All rights reserved.

Figure 1.2 Signing in to the App Hub website

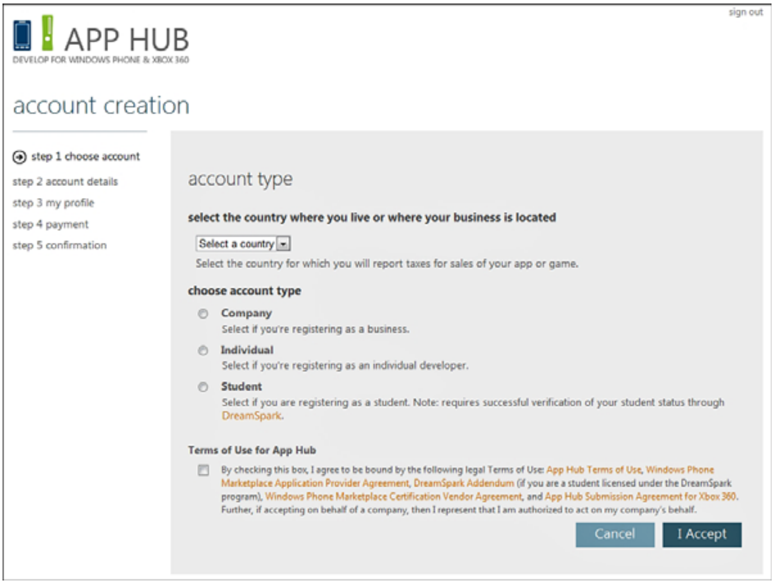
After signing into the App Hub for the first time you will be asked to select a country and an account type of company, individual, or a student. Select the option that best represents you and click the I Accept button (Figure 1.3).

The next screen will ask for your personal details such as name and address (Figure 1.4). Enter your information and click the Next button.

The next page allows you to select an image to represent your profile in the forums and an Xbox Gamertag if you don't have one already (Figure 1.5). After making your selection press the Next button.

Finally you will need to select a membership and add your payment information (Figure 1.6). A membership currently costs $99 a year.

After you have completed the payment process your membership is complete and you are ready to go.


sign out
APP HUB
DEVELOP FOR WINDOWS PHONE & XBOX 360
step 1 choose account
step 2 account details
step 3 my profile
step 4 payment
step 5 confirmation
Cancel
I Accept

account creation

account type

select the country where you live or where your business is located

Select a country

Select the country for which you will report taxes for sales of your app or game.

choose account type

○ **Company**
Select if you're registering as a business.

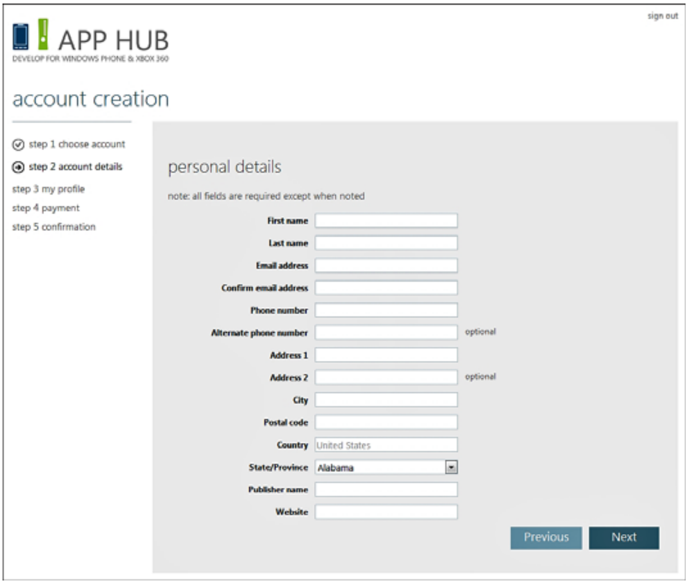
○ **Individual**
Select if you're registering as an individual developer.

○ **Student**
Select if you are registering as a student. Note: requires successful verification of your student status through DreamSpark.

Terms of Use for App Hub

☐ By checking this box, I agree to be bound by the following legal Terms of Use: App Hub Terms of Use, Windows Phone Marketplace Application Provider Agreement, DreamSpark Addendum (if you are a student licensed under the DreamSpark program), Windows Phone Marketplace Certification Vendor Agreement, and App Hub Submission Agreement for Xbox 360. Further, if accepting on behalf of a company, then I represent that I am authorized to act on my company's behalf.

Figure 1.3 Selecting an account type


sign out
APP HUB
DEVELOP FOR WINDOWS PHONE & XBOX 360
step 1 choose account
step 2 account details
step 3 my profile
step 4 payment
step 5 confirmation
Previous
Next

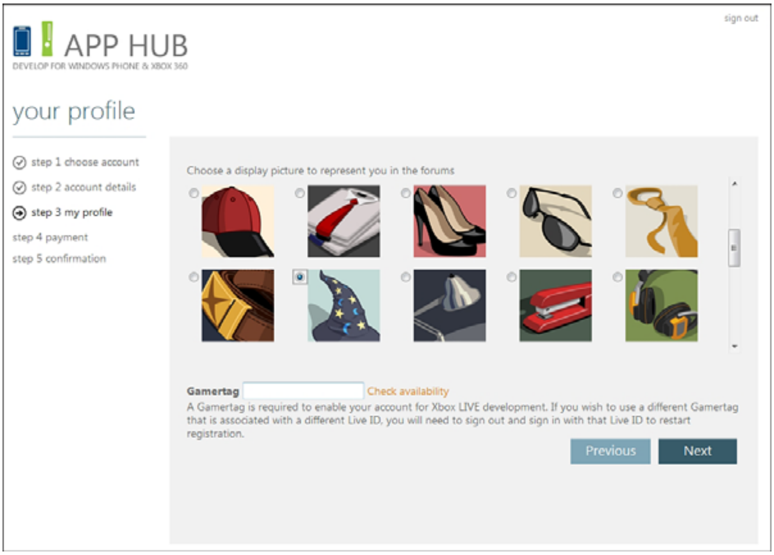
account creation

personal details

note: all fields are required except when noted

First name		
Last name		
Email address		
Confirm email address		
Phone number		
Alternate phone number		optional
Address 1		
Address 2		optional
City		
Postal code		
Country	United States	
State/Province	Alabama	
Publisher name		
Website		

Figure 1.4 Personal details page

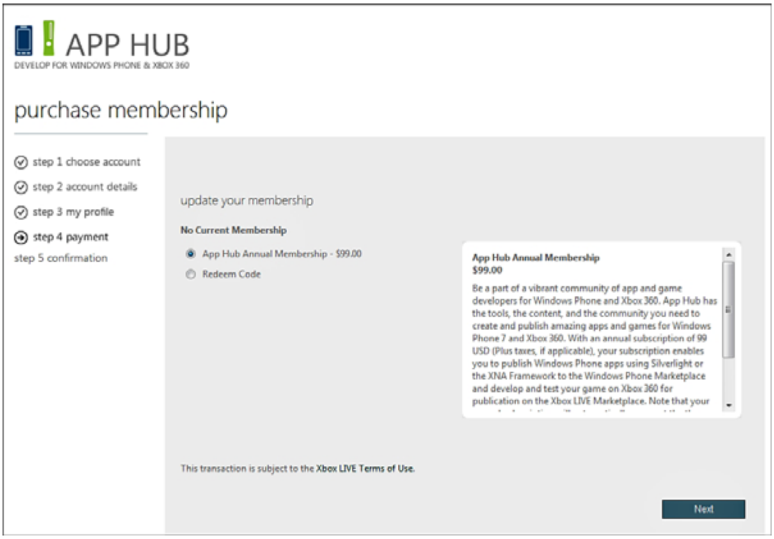

sign out
APP HUB
DEVELOP FOR WINDOWS PHONE & XBOX 360
your profile
step 1 choose account
step 2 account details
step 3 my profile
step 4 payment
step 5 confirmation
Check availability
Previous
Next

Choose a display picture to represent you in the forums

Gamertag

A Gamertag is required to enable your account for Xbox LIVE development. If you wish to use a different Gamertag that is associated with a different Live ID, you will need to sign out and sign in with that Live ID to restart registration.

Figure 1.5 Selecting personal image and gamertag


APP HUB
DEVELOP FOR WINDOWS PHONE & XBOX 360
purchase membership
step 1 choose account
step 2 account details
step 3 my profile
step 4 payment
step 5 confirmation
Next

update your membership

No Current Membership

● App Hub Annual Membership - $99.00

○ Redeem Code

App Hub Annual Membership
$99.00

Be a part of a vibrant community of app and game developers for Windows Phone and Xbox 360. App Hub has the tools, the content, and the community you need to create and publish amazing apps and games for Windows Phone 7 and Xbox 360. With an annual subscription of 99 USD (Plus taxes, if applicable), your subscription enables you to publish Windows Phone apps using Silverlight or the XNA Framework to the Windows Phone Marketplace and develop and test your game on Xbox 360 for publication on the Xbox LIVE Marketplace. Note that your

This transaction is subject to the **Xbox LIVE Terms of Use.**

Figure 1.6 Selecting membership

XNA Game Studio Connect

When you develop your games for the Xbox 360, you write the code in Visual Studio on your Windows PC and then send that code to your Xbox 360 where it runs. You still have the ability to debug in Visual Studio from your PC the code that runs on the Xbox 360.

To connect and send your code over to the Xbox 360, your Xbox needs to run the XNA Game Studio Connect title. You need to download XNA Game Studio Connect from the Xbox LIVE marketplace. Go to the marketplace on your Xbox 360 and select All Games, select XNA Creators Club, and then select the XNA Game Studio Connect.

After the download has finished, you need to launch XNA Game Studio Connect by going to your games and selecting All Games. Scroll down to the bottom of the list and launch XNA Game Studio Connect.

If you see an error message that says you need an XNA Creators Club Premium membership when you launch the XNA Game Studio Connect, this is because the account that is currently logged in does not have the membership and can't run XNA Game Studio Connect.

XNA Game Studio Device Center

After you launch XNA Game Studio Connect for the first time, notice the 5-by-5 code at the bottom of the screen. This is used to connect your Xbox 360 to your Windows PC that you use for development.

To connect these devices, you need to launch the XNA Game Studio Device Center application. You can find XNA Game Studio Device Center by going to the Start menu, selecting All Programs, selecting Microsoft XNA Game Studio 4.0, and then clicking XNA Game Studio Device Center.

> **Note**
>
> Your Xbox 360 needs to be on the same local area network subnet to connect to your Windows PC.

> **Note**
>
> You see lower deployment and debugging performance if your Xbox is connected over Wi-Fi.

Next, click the Add Device button and select the Xbox 360 as the type of device you want to add. You are prompted to enter the 5-by-5 code that is displayed on the Xbox 360 in XNA Game Studio Connect. After entering a valid 5-by-5 code, your Xbox 360 is connected to your Windows PC. You can close the XNA Game Studio Device Center.

> **Note**
>
> You might notice that the Zune is listed as a device you can add. XNA Game Studio 4.0 does not support developing for the Zune, but past versions did and the option remains to support past releases.

Your Xbox 360 is now ready for development.

Windows Phone Developer Registration Tool

Although you can run your Windows Phone 7 games in the emulator without any additional setup, it is feels amazing to see your code run on a real device. To develop games or applications for your Windows Phone 7 device, you first need to unlock your phone.

To setup your Windows Phone 7 device for development and debugging first plug your phone into your Windows PC using the cable provided with your phone. The Zune client software on your PC should start by default, but if it does not, you need to start the Zune client software.

To register your phone, you need to use the Windows Phone Developer Registration tool that can be found under the Windows Phone Developer Tools Start menu. When you launch the Windows Phone Developer Registration tool, you will see a place to enter you username and password for the account that you created previously. After your credentials are verified, your phone is ready for development and debugging.

Writing Your First Game

Now that you have installed the tools and set up your Xbox 360 and Windows Phone 7 device, you are ready to create your first game. The first game we create is the default template for the three different game types. When you create a new project, a new class inherits from `Microsoft.Xna.Framework.Game`. This is your game class and overrides some important methods that you will use. For example, a `LoadContent` method loads all of the game content. An `Update` method run each frame is where you should handle updating objects and reading user input. A `Draw` method, by default, clears the screen to a solid color called `CornflowerBlue`.

Your First XNA Game Studio Windows Game

To create your first XNA Game Studio Windows Game, you need to first open Visual Studio. To create the new project, select File, and then select New Project. In the left column, be sure to select XNA Game Studio 4.0, which is under the Visual C# tree. You can see the number of projects that you can create. Select the Windows Game (4.0) project, give it a name, and click the OK button.

The new project opens to the generated new `Game` class. You can now run the game by pressing the F5 key. You should see a solid light blue window display. You can press the Escape key or click the Close button on the window to exit the game.

That's all there is to it. You have created your first XNA Game Studio Windows game.

Your First XNA Game Studio XNA Xbox 360 Game

Now let's create your first XNA Game Studio Xbox 360 game. Just like before, select the File, New Project menu in Visual Studio. Select the Xbox 360 Game (4.0) project, give it a name, and click OK.

The new project opens to the `Game` class. If you have XNA Game Studio Connect running, you can start the game by pressing F5.

Again, you can see your game run on the Xbox 360, a light blue screen. Pressing the Back button on the controller exits the game back to XNA Game Studio Connect.

You have now built your first XNA Game Studio Xbox 360 game and run it in on the console.

Your First XNA Game Studio Windows Phone 7 Game

Finally, let's create a Windows Phone game. Go to File, New Project in Visual Studio. In the New Project dialog, select the Windows Phone Game (4.0) project, give it a name, and click the OK button.

You can run your Windows Phone games on both the Windows Phone 7 Emulator and on the physical Windows Phone 7 device. There is a drop-down selector in the upper left of the tool bar in Visual Studio that enables you to select which to use.

Select Windows Phone 7 Emulator from the drop-down menu and press F5. This launches the Windows Phone 7 Emulator and starts your game. Pressing the Back button on the emulator exits your game but the emulator continues to run. You can leave the emulator running between debugging sessions to speed up your development process.

Now select Windows Phone 7 Device from the drop-down menu in Visual Studio and press F5. If you have your Windows Phone 7 device connected and registered, the game launches on your device showing the light blue screen. Pressing the hardware Back button exits your game.

Download Samples

The samples in this book can be downloaded from the following URL:

http://www.informit.com/title/9780672333453

There are also many educational resources available on the App Hub site at the following URL:

http://create.msdn.com

Summary

Congratulations, you have in a short period of time created games that run on three different platforms including Windows, Xbox 360, and Windows Phone 7. As you continue to read this book, you will learn much more about how to create a compelling game that uses 2D and 3D graphics, audio, input, storage, media, avatars, gamer services, and networking.

Now that you have your development PC, Xbox 360, and Windows Phone 7 device set up for development, you can start to create some games. Don't forget that you can download all of the samples from this book at http://www.informit.com/title/9780672333453.

In the next chapter, we start to draw and animate 2D images to the screen.

2

Sprites and 2D Graphics

Now that you have had a brief introduction to what Game Studio is and how to get started, you might as well dive right in and start showing awesome things on screen! Before we get into the more complicated three-dimensional (3D) visuals you see later in this book, we first start with more simple two-dimensional (2D) graphics. In this chapter, you learn the following:

- What 2D means
- Rendering something on screen
- Using the Spritebatch object
- Rendering text

What Does 2D Mean?

What exactly does 2D mean? You can probably guess that it is short for two-dimensional, but what does that mean? Almost all games that you see have visuals on a monitor, a television, or something else that is inherently flat and two-dimensional. For the purposes of this book, when we say 2D, we mean that things are rendered using sprites in screen coordinates.

Of course, this begs the question, "What are screen coordinates?" Look at your monitor because that is where the "screen" in "screen coordinates" comes from after all. The upper, left pixel of your monitor is the screen coordinate of 0,0 (which is sometimes called the origin). In a 2D coordinate system, the first number is the X axis, whereas the second number is the Y axis, and in the case of your monitor, the X increases as you move to the right, and the Y increases as you move down. For example, if you had a resolution of 1280×720, the lower, right pixel of your monitor is at a screen coordinate of 1279,719. This means that the top, right pixel is a screen coordinate of 1279,0, whereas the bottom, left pixel is at 0,719, as seen in Figure 2.1.

Going back in time, you could argue that some of the earliest 3D games fit this description. After all, games such as the original Doom, Wolfenstein, and Duke Nukem 3D all claimed to be 3D, yet they were nothing more than sprites rendered onto the screen that gave the illusion of 3D. Back then, those games were referred to as 2.5D; recently, the term has been repurposed to mean games that are fully rendered in 3D but where game play happens solely in 2D.

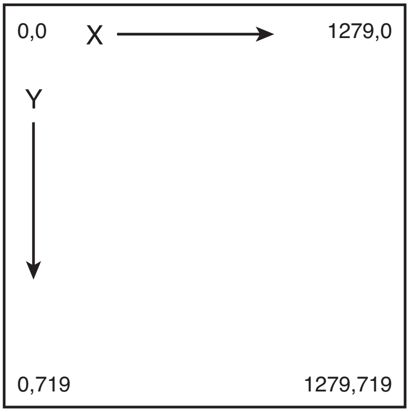


Figure 2.1 2D screen coordinates

Another early term you might not have heard of is sprite. What exactly is a sprite (aside from the soda)? In simplest terms, it is a 2D image that you render on screen. All images you see on the Web, for example, are sprites. At its simplest level, creating a 2D game is nothing more than drawing a lot of pictures on screen at once.

Show Me Something on Screen

In Chapter 1, "Getting Started," you spent some time creating projects and saw them run with nothing showing except a solid colored background. Let's expand on those projects and show new and hopefully interesting things instead! Start by creating a new game project in Visual Studio and add a new variable to the list of variables the project already has defined, as shown in the following:

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
Texture2D texture;
```

`Texture2D` is a class that holds any images you might want to render on screen. In later chapters, you see how this is also used to give detail to 3D models, but for now, it is simply a data store for the image we want to show in our game. Also notice the other two variables declared by default, namely the `SpriteBatch` and `GraphicsDeviceManager`.

The `GraphicsDeviceManager` is discussed in more detail in later chapters, but the `SpriteBatch` object is what you use to do the actual rendering. This object is discussed in depth throughout this chapter, but before you learn about the intricacies of rendering 2D objects, first, you shouls just get something showing on the screen.

You need an image to draw to the screen. In your Visual Studio solution, notice that you have two projects: your code project where your game is and a content project next to it. This content project is where you add the images you want to draw and it is discussed in more detail in later chapters. Right-click your content project now, and then choose Add->New Item. Feel free to pick an image for your computer (ensure the image is a common image format such as jpg, bmp, png, or gif), or use one from the downloadable examples. The example included with the downloadable examples uses the file glacier.jpg, which is used in some of the samples you can download for Game Studio 4.0.

> ### Content Item Properties
>
> Selecting an item in a content project (such as the image you just added) and viewing its properties show you many different options depending on the type of item you have selected. For now, the only one that is important is Asset Name, which, by default, is the name of the file you've added to the content project without the extension. In the case of the code in the downloadable examples, it is glacier. The other properties are explained in more detail in later chapters.

Getting something on screen is remarkably easy from here. First, you need to scroll through your project's `game1.cs` file and find the `LoadContent` method. As the name implies, this is where you load the content for your game, so add the following line of code to that method:

```
texture = Content.Load<Texture2D>("glacier");
```

If you used a different image other than the glacier image this example uses, you need to enter that filename (without the extension) instead. This takes the image data from your picture and loads it into the `Texture2D` object you specified. The `Content` object is an instance of the `ContentManager`, which was automatically created by the new project. Again, the content manager is discussed in more depth later, but for now, it is the thing that loads your content.

All that is left is to draw your image on the screen. Look through your project to find the `Draw` method and replace the method body with the following code:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
spriteBatch.Begin();
spriteBatch.Draw(texture, GraphicsDevice.Viewport.Bounds, Color.White);
spriteBatch.End();
base.Draw(gameTime);
```

The first and last lines are the same as what is already in your project, whereas the middle three lines are where the actual drawing of your image is. Run the project and you should see an image similar to the one shown in Figure 2.2 if you used the glacier image provided with the downloadable examples or an image you chose that covered the entire window if you did not use the glacier image.

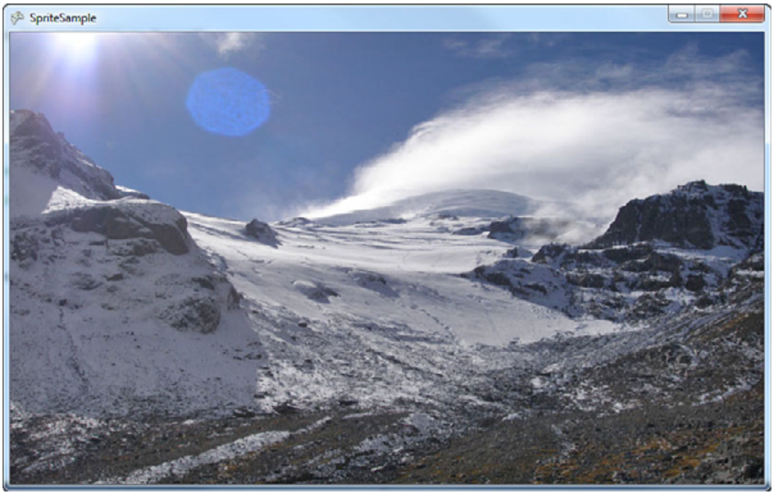


Figure 2.2 A 2D image drawn on screen

A theme you will notice throughout this book is how easy it is to do simple operations using Game Studio 4.0. This is an example of this simplicity. With a mere five lines of code, you've done more work than you probably realize, rendering your image on to the screen. With the instant gratification of seeing something, now it's time to take a step back and see what has gone into rendering this picture.

Spritebatch

When you first create a new project, a `SpriteBatch` object is declared and instantiated in the `LoadContent` method. This is the main object used to render 2D graphics, and virtually all games (even full 3D games) need to render 2D graphics at some time. This object can be used to draw a single sprite like you just did, or it can draw many sprites with a wide variety of options that can be used to control exactly how you want your images drawn.

Drawing

First, let's look at the `Draw` method, which is the one that actually got the picture on to the screen:

```
spriteBatch.Draw(texture, GraphicsDevice.Viewport.Bounds, Color.White);
```

This is only one overload of the `Draw` method (there are seven total), but it is one of the simplest. The first parameter is obvious—what texture do you want to draw? In the project earlier, you loaded your image into this texture, so that is what is rendered on the screen. This is one of the two parameters required and it is in every overload for the `Draw` method.

The other nonoptional parameter to `Draw` is the last one in the call, the color. This is the color your image is "tinted" with (in actuality, it is the color of each pixel in your image multiplied by the color specified, which is discussed later). Passing in `Color.White`, as done here, causes your image to appear with the same colors as the original image, whereas passing in `Color.Black` causes the image to be replaced by solid black. If you change this to `Color.Red`, and you will notice that the image is now tinted red. In many cases, you can simply leave this `Color.White`.

The middle parameter in the `Draw` call is the destination rectangle. As the name implies, it is the rectangle where you want the drawing to occur (hence, the destination). This rectangle is specified in screen coordinates where 0,0 is the upper left corner of the rendering area, which in this case is the window. When in full-screen mode, it is the upper left corner of the monitor as mentioned earlier. The destination rectangle is useful because if the image you are drawing isn't the exact size of the rectangle you're drawing, it automatically stretches or shrinks to fit exactly in that area.

In this case, the destination rectangle is calculated using a property of the `GraphicsDevice` object. The `GraphicsDevice` is the object that encapsulates the portions of the graphics card you can control, whereas the `Viewport` property contains information about where the device renders to on the screen. You used the `Bounds` property because it returns the rectangle that encompasses the entire rendering area, which is why the image you chose covers the entire window, regardless of what size it was originally when it was loaded.

> **Note**
>
> The GraphicsDevice object and its properties are discussed in depth in Chapter 3, "The Game Object and the Default Game Loop."

Before we look at other methods used on the sprite batch, let's look at the more overloads for drawing images on the screen. In your project, right-click the content project and add a reference to `cat.jpg`, which you can find in the downloadable examples. Anyone familiar with many of the samples that are available for Game Studio realizes that cats appear all over the samples. This one, however, is a different cat—it is my own. He is a little bit camera shy though, so please be gentle with him.

Change the code that loaded the previous image to load the cat image, as the following:

```
texture = Content.Load<Texture2D>("cat");
```

This image is 256×256 pixels, and your window by default is 800×480 pixels. Running the project right now shows you a stretched version of the cat covering the entire rendering surface, much like you saw earlier. Now change the `Draw` method to the following:

```
spriteBatch.Draw(texture, Vector2.Zero, Color.White);
```

Notice that a different parameter of type `Vector2 replaced` the destination rectangle. Like the name implies, `Vector2.Zero` returns a vector with both the X and Y value as zero. The new parameter, called the position, tells the sprite batch where to begin drawing, so the upper left pixel of the image draws at the spot specified by this parameter (in this case 0,0, or the upper, left corner of the rendering area).

> **Note**
>
> The `Vector2` class is a value type used to store two-dimensional coordinates, namely an X and Y value.

Using this overload and specifying a position means that no stretching of the image occurs; it renders whatever size the source image actually is. Run the application and notice that the cat is now in the upper, left corner of the window, but there are large areas of empty background everywhere else. To render the cat in the center of the window, modify the `Draw` call as follows:

```
spriteBatch.Draw(texture,
new Vector2((GraphicsDevice.Viewport.Width-texture.Width)/2,
    (GraphicsDevice.Viewport.Height - texture.Height) / 2), Color.White);
```

This combines information from the `Viewport`, along with information about the texture to properly position the image in the center of the window as in Figure 2.3. It still isn't exciting though because it is a static image that doesn't do anything.

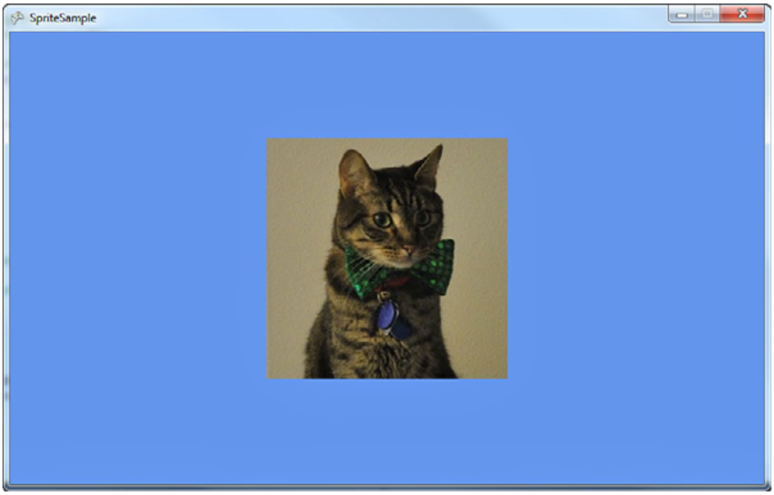


Figure 2.3 The cat centered on the screen

Notice that the two overloads so far do remarkably similar things. They both draw an image to a particular place on the screen of a particular size. The only difference between them is the one with the destination rectangle requires you to specify a width, a height, and a position. As a matter of fact, these three lines produce the same results.

```
spriteBatch.Draw(texture, Vector2.Zero, Color.White);
spriteBatch.Draw(texture, new Rectangle(0,
0, texture.Width, texture.Height), Color.White);
```

Moving Things Around

You have much more control over the rendering than this, however. Change your `Draw` call to the following and run the application:

```
spriteBatch.Draw(texture, new Vector2(100,100),
    null, Color.White, 0.3f, Vector2.Zero, 1.0f,
SpriteEffects.None, 1.0f);
```

Notice that the cat is drawn with the upper, left corner at 100,100 as you specified in the position vector, but it is now rotated slightly. This overload is more complex than the previous ones, so let's look at the new parameters individually.

The first new parameter is the third one, which is listed as null. This is the source rectangle, and unlike the destination rectangle, it controls what you draw rather than where you draw (discussed more in depth later in the chapter). Directly after the tint color is type float, which is the rotation angle you want to render at. This angle is specified in radians.

> **Note**
>
> To convert angles in degrees to radians, use the `MathHelper.ToRadians` method.

The next parameter is the origin (another vector), which we discuss in a moment. Directly after the origin is the scale parameter, which you use to uniformly scale the image. A scale of 1.0f is normal size, 2.0f is twice the size, and 0.5f is half the size.

Next up is the `SpriteEffects` enumeration, which has three options: `None`, `FlipHorizontally`, and `FlipVertically`. Each of these do exactly as the name implies. Passing in `None` does no special processing of the image, whereas passing in either of the other two flips the image before drawing it, either horizontally or vertically. For example, if you use `SpriteEffects.FlipVertically`, the cat is drawn upside down.

The final parameter is called the layer depth. This value should be between 0.0f and 1.0f with 1.0f is "on top" and 0.0f is "on bottom." This enables you to control how the images are sorted when you draw more than one, and it is ignored unless the sort mode is set to either `BackToFront` or `FrontToBack`. We talk about sort modes later in the chapter.

This is the largest overload, and it has every feature you need in drawing a sprite. There is another overload that has the same number of parameters, but replaces the single scale parameter with a `Vector2`. This enables you to scale your image with different scaling values for the X and Y axis. So if you passed in a scale vector of 2.0f, 1.0f, the image would be twice as wide, but the same height. A scale vector of 0.5f, 2.0f causes the image to be half as wide, but twice the height.

Now it is time to go back to the mysterious origin parameter. The origin is the point around which the rotation occurs. In the previous example, you used `Vector2.Zero`, so your rotation is around the upper left corner of the image. If you use `new Vector2(texture.Width/2, texture.Height/2)` instead, the image rotates around the center (see Figure 2.4).

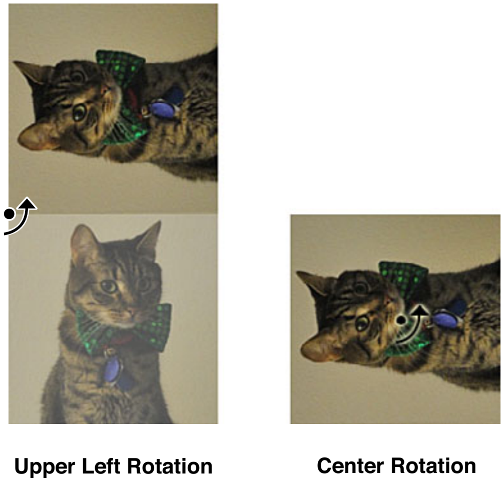


Figure 2.4 Rotation origin

Animation

There is only one parameter to the `Draw` overloads left to discuss, which is the source rectangle. As mentioned, the source rectangle lets you control the portion of the image you draw. Up until now, you drew the entire image, which is the default behavior if this parameter is not specified or if it is null. What if you had a single image that had multiple smaller images inside it, and you only wanted to draw a portion of it? That is what the source rectangle is for. The cat image is 256×256 pixels, but if you specified a source rectangle of (0,0,256,128), it renders the top portion of the cat and not the bottom portion.

One common usage of the source rectangle is for simple 2D animations. Your image contains several "frames" of animation that you swap through in succession to give the illusion of motion. Cartoons have been doing this for years, and it is a common technique.

In your content project, add another existing item and this time, include the spriteanimation.png image from the downloadable examples. Like before, update the `LoadContent` method to change which texture you load.

```
texture = Content.Load<Texture2D>("spriteanimation");
```

If you run the project now, you can see a weird image. It looks like there are several images instead of just a single one. That is exactly what the image is—a lot of smaller images stored in a larger one. In this case, there are ten separate 96×96 images stored within a single 960×96 image. If you change the `Draw` call in the `Draw` method as in the following, a single image is now drawn:

```
spriteBatch.Draw(texture, new Vector2(100, 100),
    new Rectangle(0, 0, 96, 96), Color.White);
```

The source rectangle of 0,0,96,96 tells the sprite batch to render only from the first smaller image within the file and to ignore the rest. By itself, though, it is still bland. Replace the `Draw` call with the following code to see the animation play out:

```
int frame = (int)(gameTime.TotalGameTime.TotalSeconds * 20) % 10;
spriteBatch.Draw(texture, new Vector2(100, 100),
    new Rectangle(frame * 96, 0, 96, 96), Color.White);
```

Now you see a guy running in place at a position of 100,100! What you've done here is select which portion of the source image to draw from based on the current amount of time the game has run. You've taken the total number of seconds the game has run (this is a fractional value, so it records portions of a second, too), and multiplied it by 20, which forces the animation to run 20 times per second. You then pick the current frame by doing a modulus against the total number of frames and some quick math to pick the correct source rectangle based on that frame.

This technique is a common way 2D games render animations.

Controlling State

With all of the possible parameters of the `Draw` overloads discussed, now it is time to look at the multiple overloads of the `Begin` method. Although we use this method in every example up until this point, you used only the overload with no parameters, and that doesn't require much explanation. Each example up until now has had only a single `Draw` call, but as the name sprite batch implies, you can draw many images at once, and you need some way to control how each image interacts with every other. This is what the `Begin` overloads do.

Before we get into that, however, add an existing item to your content project and include the image layers.jpg from the downloadable examples. This image has a few pictures and numbers on it to help better demonstrate the behavior of multiple `Draw` calls in a single batch. Update your load content method as always when you add new content to your project:

```
texture = Content.Load<Texture2D>("Layers");
```

Again, replace the contents of your `Draw` method with the following:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
spriteBatch.Begin(SpriteSortMode.Texture, null);
for (int i = 0; i < 4; i++)
{

Rectangle src = new Rectangle((i % 2) * (texture.Width / 2),
        (i < 2) ? 0: (texture.Height / 2) ,
        texture.Width/2, texture.Height/2);

    spriteBatch.Draw(texture, new Vector2(50 + (50*i), 50 + (50*i)), src,

Color.White, 0.0f, Vector2.Zero, 1.0f, SpriteEffects.None, i * 0.1f);
}
spriteBatch.End();
base.Draw(gameTime);
```

This uses the largest overload of the `Draw` method and renders four different squares of the texture each at a different spot on the screen. Notice also the layer depth is passed so that each image is rendered at a different depth with the image containing the number 1 drawn at a depth of 0.1f and with the image containing the number 4 drawn at a depth of 0.4f. Also notice that this code renders the images in the order of 1,2,3,4. However, when you run the code, it is not drawn like that—it draws seemingly random, much like in Figure 2.5.

However, if you change the call to `Begin` with the following, it draws the images with the first image on the bottom and the last image on top, much like you see in Figure 2.6.

```
spriteBatch.Begin(SpriteSortMode.FrontToBack, null);
```

This is because this overload of the `Begin` call includes the first parameter of type `SpriteSortMode`, which controls how multiple sprites sort within this batch before drawn on the screen. The options for this enumeration include `FrontToBack` as seen here, which renders images with the highest layer depth "on top" and the lowest layer depth "on bottom." The image with the 4 on it is rendered on top because it has the highest layer depth. If you instead switched this to `BackToFront`, the order reverses itself, and the lowest layer depth is "on top," and in this case, the image with the 1 is rendered on the top.

Another option for sorting is the default sorting option `Texture`. This causes the `Draw` calls to sort by the texture. In the previous example code, all of the `Draw` calls use the same texture, so there is no special sorting. This is the default because drawing a few images, switching textures, drawing a few more images, switching back to the original texture, and drawing a few more images can hinder performance.

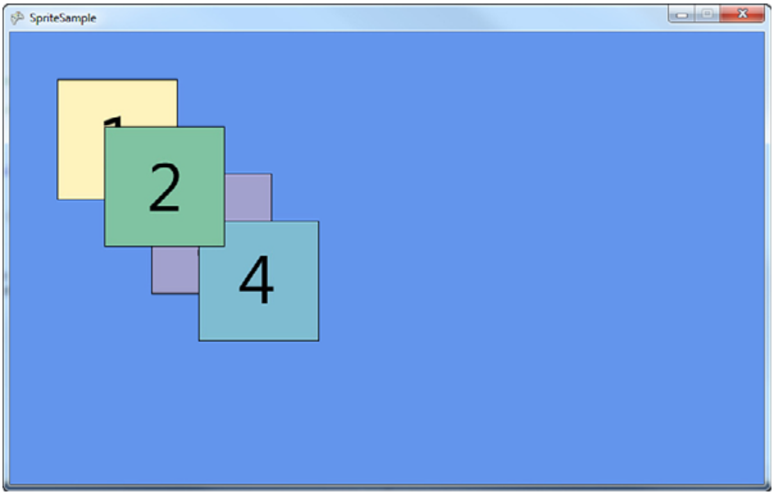


Figure 2.5 Rendering multiple sprites with no control

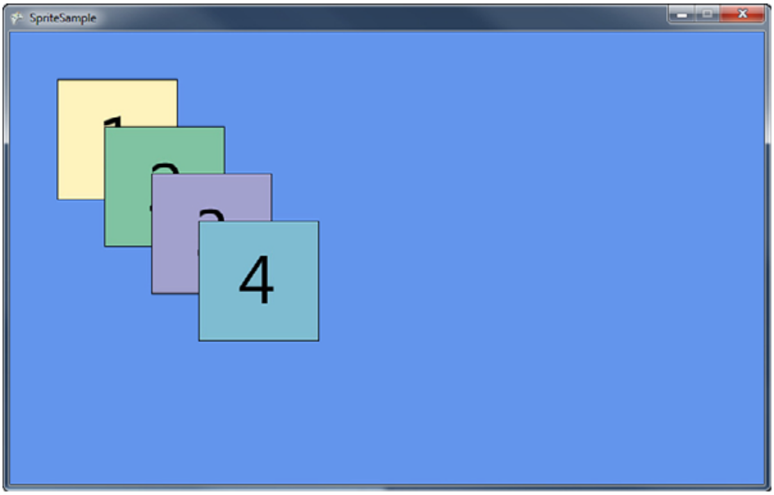


Figure 2.6 Rendering multiple sprites with layer control

Other options for the sorting mode include `Deferred`. Each image sorts in the order it is used in the `Draw` calls. This means that all `Draw` calls should be "batched up" to make as few actual rendering calls to the hardware as possible. This is because an actual rendering call on the hardware can be expensive, and it is better to make fewer rendering calls that render lots of data instead of a large amount of render calls that render small amounts of data. All options in the sorting mode except for one also infers the behavior of `Deferred`.

The last sort mode is the one that doesn't follow the `Deferred` behavior and is called `Immediate`. This tells the sprite batch to make a call to the rendering hardware for every call to `Draw` you make.

The other parameter to `Begin` shown previously and that is currently null is the `BlendState` you want to use for this sprite batch. Although we discuss the blend state (and the other states) in later chapters, now is a good time to understand the basics of blending. As the name implies, blending controls how multiple images are combined. The default value (what is used if you pass in null) is `BlendState.AlphaBlend`. The alpha values control the transparency of objects drawn, so this blending state tells the sprite batch that for every pixel it draws, it should render the top-most pixel if the pixels are opaque (have no alpha value), or it should blend the top most pixel with the pixels "underneath" it.

In your content project, add another existing item, AlphaSprite.png. This item is a png (much like you did when you picked the animated sprite), because this file format can include alpha data. Instead of using the same texture object you've been using, though, add a new texture variable to your class so you can see how things blend together:

```
Texture2D alphaTexture;
```

Of course, you need to load this texture, so add that to the `LoadContent` method:

```
alphaTexture = Content.Load<Texture2D>("AlphaSprite");
```

Then replace the `Draw` method with the following:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend);
spriteBatch.Draw(texture, Vector2.Zero, Color.White);
spriteBatch.Draw(alphaTexture, Vector2.Zero, Color.White);
spriteBatch.End();
base.Draw(gameTime);
```

This renders the full four-picture sprite you used previously along with a little character in the middle of it (see Figure 2.7).

Now change the blend state from alpha blending to `BlendState.Opaque`, which doesn't attempt to blend the two images. Because of the `Deferred` sort mode, it renders them in the order they appear, so it renders the character last. Notice that you can't see anything other than the character, and the portions that used to be transparent are now black. Change it again to `BlendState.Additive` and notice a weird combination of the two images that look too "bright." This is caused by "adding" the colors together.

Note

Blending is discussed in depth in Chapter 7, "Blending, States, and Textures."

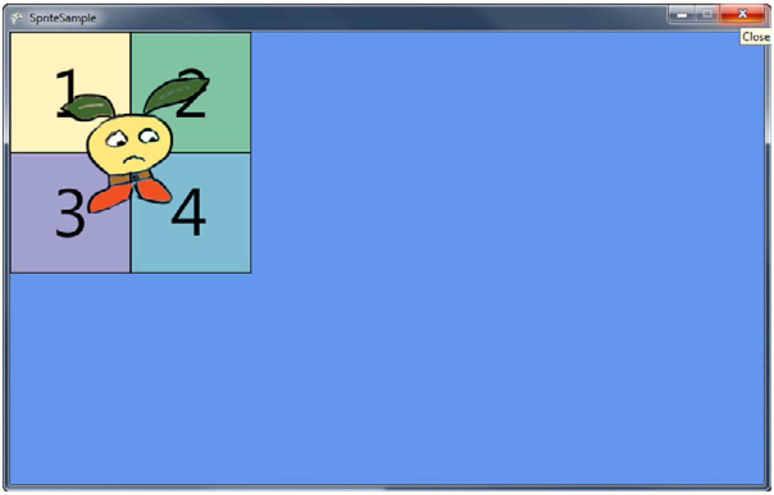


Figure 2.7 Rendering multiple sprites with alpha blending

There are other parameters to other overloads of the `Begin` method, including the depth stencil state, sampler states, rasterizer state, which effect is used, and a matrix for transformations. Each of these is discussed in Chapter 7.

Rendering Text

Another important reason to use the sprite batch is to render text to the screen. Notice a `DrawString` method on the sprite batch object, which is the method you use to draw the text. Before you get there, you need to do a few extra steps.

First, add a new item to your content project by right-clicking the project and selecting Add->New Item. In the dialog box, select the option to add a Sprite Font, name it Game, and click the Add button. Visual Studio opens this file, which is simply a standard XML file.

The first node of note is the `FontName` node, which (as expected) is the name of the font. The default here is Kootenay although we personally like Comic Sans MS. Select a font (browse through your fonts folder), and type the font name in the node. The next node is `Size`, which is how large you want the font to be. The default here is 14, but you can choose any size you'd like. The size roughly matches what you see if you change the font size in a text editor such as notepad. The sample with the downloadable examples is a size of 48.

Next is `Spacing`, which enables you to control how the letters are laid out when they draw. The default value is zero, so no extra spacing is placed between characters as they draw, but changing the value adds extra space (in pixels) between characters. It's even possible to use a negative number to remove space between characters. Hand in hand with spacing, the next node is called `UseKerning` and controls whether kerning is used when rendering the text. Kerning changes how the spacing between characters is laid out. Rather than a fixed amount of space between each character, the amount of space depends on the character that comes before and the character that comes after. The default is true because in most cases it just looks better that way.

Next is the `Style` node, which enables you to choose what the style of the font. The default is `Regular`, but you can also choose `Bold`, `Italic`, or if you are adventurous, choose `Bold, Italic`. Note that these styles are case sensitive.

The last set of nodes controls which characters you expect to draw. The default includes all English characters (ASCII codes 32 through 126). To draw characters beyond this range, you can update the values here.

Now it's time to get some text on the screen. First, add a new variable to the class to hold the font data:

```
SpriteFont font;
```

Then, load the new font in your `LoadContent` method:

```
font = Content.Load<SpriteFont>("Game");
```

Finally, replace your `Draw` method with the following to see how easy it is to add text rendering to your games:

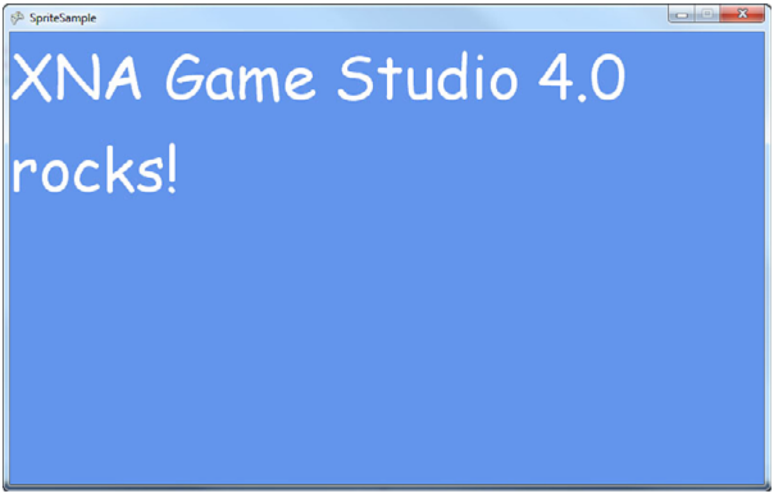
```
GraphicsDevice.Clear(Color.CornflowerBlue);
spriteBatch.Begin();
spriteBatch.DrawString(font,"XNA Game Studio 4.0", Vector2.Zero, Color.White);
spriteBatch.End();
base.Draw(gameTime);
```

Everything except the `DrawString` should be quite familiar by now. There are a number of `DrawString` overloads, many of which mirror ones you saw earlier for rendering images rather than text. The `DrawString` methods do not take a `Texture2D`, but instead take a `SpriteFont` to get the font data. They also include an extra parameter that is either a `string` or a `StringBuilder` that is the text you want to render. They do not, however, include a destination rectangle or a source rectangle (because these do not make sense when rendering text). All other parameters to `Draw` exist in the overloads of `DrawString`, however, so you can freely scale, rotate, color, and layer your text rendering.

The sprite font object has a few methods and properties that are somewhat interesting. For example, what if you wanted to draw a second line of text just below the line you've already drawn? How would you know where to begin? Add the following lines of codes to your `Draw` method directly below the `DrawString` call:

```
Vector2 stringSize = font.MeasureString("XNA Game Studio 4.0");
spriteBatch.DrawString(font, "rocks!", new Vector2(0,stringSize.Y),
➥Color.White);
```

Here, you use the `MeasureString` method on the sprite font object, which calculates the size (in pixels) that the rendered string takes up. You can then use that size to start the next line of text rendered down by an appropriate number of pixels. Running the application now shows you two lines of text as in Figure 2.8.



XNA Game Studio 4.0

rocks!

Figure 2.8 Text rendering

Summary

In this chapter, you learned the basics of simply showing an image onscreen along with rendering animated characters, rotating images, blending between multiple images, and drawing text on the screen.

Before moving on to more rendering techniques, you should get a basic understanding of what the rest of the projects you created are doing and the basic game loop.

This page intentionally left blank

3

The Game Object and the Default Game Loop

When you create a new project, many things happen behind the scenes, and many features to help drive the game available to you. In this chapter, you learn about these features, including:

- The game class
- The standard game loop
- Game components

Up until now, you've created new projects and added some code to do other fancy things, but you haven't taken a step back to look at the default game template. Now is a good time to take a look at what is provided for you automatically and the other features available.

What Is in a New Project?

Open Visual Studio and create a new Game Studio 4.0 Windows Game project. Notice that your main project includes two code files (program.cs and game1.cs), and you have a content project you previously used. You can safely ignore everything in program.cs because it is simply the stub that launches the game. As a matter of fact, this isn't even used on Windows Phone 7.

The interesting things that are discussed in this chapter are in game1.cs. Notice first that the `Game1` class that is created comes from the `Game` object provided by Game Studio. The initial starting project gives you everything you need to start creating a game. It has a spot for initialization, a spot to load the content your game needs, a spot to update the game state, and a spot to render everything.

More things happen behind the scenes than you are probably aware of, however. Start with the first thing you see in the constructor, creating the `GraphicsDeviceManager`.

```
graphics = new GraphicsDeviceManager(this);
```

This one line of code starts a chain reaction of operations. It naturally creates a new `GraphicsDeviceManager` (which is discussed in just a moment), but it does more than that. This object implements `IGraphicsDeviceService` and `IGraphicsDeviceManager`. When you create the object, it takes the game parameter you've passed in (that is, the `this` parameter) and adds itself to the `Services` property of the game object.

> **Note**
>
> You can create and add your own services to this property (it maintains a list of services), and it is a convenient way to get game-specific services directly from the game rather than passing them around everywhere.

After the graphics device manager has been added to the services list, the actual graphics device is created when the constructor has finished executing. The default options work just fine, but you actually do have some control over the settings the device has.

Notice that quite a few different properties on this object can be used to control how the device is created or to get information about it. The first one is the `GraphicsDevice`. Right now, it hasn't been created yet, but after it has been, it can be accessed here. You most likely never need it, though, because the `GraphicsDevice` is a property of the `Game` itself.

The `GraphicsProfile` is another property you can access. Profiles are discussed in Chapter 4, "Introduction to 3D Graphics". Next is the `IsFullScreen` property that behaves differently depending on the platform you run. The default value here is false, although on Xbox 360, it doesn't matter what this is set as because you are always full screen on that platform. On Windows, setting this to true causes the device to be created in what is called full screen exclusive mode, and your rendering encompasses the entire screen. On Windows Phone 7, this controls whether the system tray bar is visible or not visible.

> **Note**
>
> Taking over the full screen in exclusive mode on Windows is not a nice thing to do without the user asking you to do so. In modern operating systems, the graphics hardware is shared nicely between games and the operating system, and forcing the operating system to yield to your game can cause behavior that some of your players may very well find annoying (the authors here included).

The next set of properties is the most commonly changed, and it includes the preferences. Because they are preferences and not requirements, the runtime attempts to use these settings, and if it cannot use them, it falls back to what it feels is the closest to what you requested. These properties are `PreferMultisampling`, `PreferredBackBufferWidth`, `PreferBackBufferHeight`, `PreferBackBufferFormat`, and `PreferDepthStencilFormat`.

The back buffer is where your content is rendered, and the sizes in these preferences (width and height), control how large that area is. On Windows, in nonfull screen mode, this also controls the size of the window. In full screen mode, it controls the resolution of the monitor when it takes exclusive control of it. On Xbox 360 and Windows Phone 7, the devices have a built-in native resolution. For Windows Phone 7, the device has a resolution of 480×800 (in portrait mode), whereas the Xbox is configurable. On each of these platforms, if you ask for a different back buffer resolution, it is scaled to the native device resolution.

Multisampling is the process used to remove what are called the "jaggies" from rendered images. These jagged edges are formed normally on the edges of objects or on lines that are not on pixel boundaries (for example, nonhorizontal or vertical lines). Multisampling blends each pixel with other pixels around it to help soften these jagged edges. It does this by rendering the image larger and blending multiple pixels down to a single pixel. Although it doesn't necessarily remove the jagged edges, it certainly can help. There is a performance cost for doing this, so this defaults to false.

The last two preferences are the formats for the back buffer and the depth stencil. Formats are used to describe how data is laid out for the final rendered image. For the back buffer, this is how the color is laid out, and the default for this is actually `SurfaceFormat.Color`. This is a 32-bit format that has 8 bits for red, green, blue, and alpha. The depth stencil buffer formats control how many bits are used for the depth buffer and stencil buffer.

> **Note**
>
> Depth buffers are discussed in more detail in Chapter 7, "States, Blending, and Textures".

The last two properties are `SupportedOrientations`, which is mainly used for Windows Phone 7, and `SynchronizeWithVerticalRetrace`. Synchronizing with the vertical retrace is a way to prevent tearing by pausing until the device is ready to render the entire screen at once. These are discussed in depth in Chapter 11, "Understanding Performance" when performance and measurement are discussed.

There are also six different events you can hook off of the `graphics` object, most of which are self-explanatory based on the names. The one interesting one is `PreparingDeviceSettings`. This event is triggered right before the device is created, and it gives you the opportunity to override any of the settings before the device is actually created. Use this only if you know the device supports the settings you request.

There are also two methods on the object, `ApplyChanges` which attempts to instantly update the device to the current settings (or create a new device if required), and `ToggleFullscreen`, which makes a windowed game full screen and a full screen game windowed during runtime. Using either of these is rarely required.

The last thing the constructor does is set the root directory of the automatically created content manager to "Content," which is where your content project places the content you add to your game. The content manager is created for you when the game is created, so you can begin using it immediately. The content manager is discussed more in depth in Chapter 9, "Using the Content Pipeline."

```
Content.RootDirectory = "Content";
```

The default template has overrides for five common methods: `Initialize`, `LoadContent`, `UnloadContent`, `Update`, and `Draw`. Although nothing happens in `Initialize` and `UnloadContent`, the other three have basic stub code. The `LoadContent` method creates the sprite batch object you almost certainly need. The `Draw` method clears the screen to the infamous CornflowerBlue color. Finally, the `Update` method adds a quick check to see if you're pressing the Back button on your controller to see if it should exit the game. We get into the flow of these methods and how they're used in just a moment, but first, let's take a look at the `Game` class itself.

> **Note**
>
> For Windows Phone 7 projects, there is another very important aspect to the game lifetime you need to understand called "Tombstoning". It is discussed in depth in Appendix D, "Dealing with Tombstoning".

The Game Class

The `Game` class is where all of the magic in your game takes place. Almost everything in your game is driven in some part by this class, and it is the root of your project. Let's dive in and see what it is made of.

Virtual Methods

The `Game` object you're class derives from has many knobs and buttons (figuratively) to help control your game and its flow. It has several virtual methods you can override to help control different features, some of which we've seen already.

`Initialize,` as you would expect, is called once just after the object is created. It is where you would do most of your initialization (aside from loading content) for your game. This is an ideal place to add new game components for example (which are discussed later in this chapter). `LoadContent` and `UnloadContent` are two other areas where you should load (or unload) your content. Content is loosely defined and can be any external data your game needs, whether it's graphics, audio, levels, XML files, and so on.

The `Update` method is where you handle updating anything your game requires, and in many games, you can do things such as handle user input. Because most games are essentially a simulation with external forces, you need a central spot where you can perform the operations to advance that simulation. Common things you'd do include moving objects around the world, physics calculations, and other simulation updates.

`Draw` probably needs no further explanation. All drawing code for each scene occurs here. There are two other drawing methods you can override: `BeginDraw` and `EndDraw`. These are called directly before and after `Draw` is called, respectively. If you override the `EndDraw` call, you need to ensure you call the `base.EndDraw` or manually call `GraphicsDevice.Present;` otherwise, you never see your scenes drawn on screen.

> **Note**
>
> `Present` is the last call made at the end of drawing and tells the graphics device, "Ok, I'm done drawing now; show it all on screen."

Much like the pre- and post-drawing methods, there are also `BeginRun` and `EndRun` methods you can override that are called before the game begins and just after the game run ends. In most cases, you do not override these, unless you are doing something such as running multiple simulations as individual game objects.

You can override the method `ShowMissingRequirementMessage`. Most people probably don't even realize it is there. By default, this does nothing on non-Windows platforms, and on Windows, it shows a message box giving you the exception detail. This enables customization if the platform you run on doesn't meet the requirements of your game, which is normally only an issue on a platform such as Windows where you can't guarantee which features it supports.

The last three methods you can override are mirrors of events you can hook. `OnActivated` is called at the same time the `Activated` event is fired, and it happens when your game becomes activated. Your game is activated once at startup, and then anytime it regains focus after it has lost focus. To mirror that, you use the `OnDeactivated` method, which is called when the `Deactivated` event is fired, and that happens when your game becomes deactivated, such as it exits or it has lost focus. On Windows and Windows Phone 7, your game can lose focus for any number of reasons (switching to a new app, for example), whereas on Xbox 360, you see this only if the guide screen displays.

Finally, the `OnExiting` method is called along with the `Exiting` event. As you can probably guess, this happens just before the game exits.

Methods

Most of the methods on the Game class are virtual so there aren't many here to discuss, and they're almost all named well, so you can guess what they do. The `Exit` method, which causes the game to start shutting down. The `ResetElapsedTime` method resets the current elapsed time, and you'll learn more about it later in this chapter. The `Run` method is what starts the game running, and this method does not return until the game exits. On Xbox 360 and Windows, this method is called by the autogenerated main method in program.cs at startup, and on Windows Phone 7, this method throws an exception. Due to platform rules, you can't have a blocking method happen during startup on Windows Phone 7. A timer starts and periodically calls `RunOneFrame`, which does the work of a single frame. You can use this method on Xbox 360 and Windows, but you shouldn't have to use it since the game object is doing that for you.

The `SuppressDraw` method stops calling `Draw` until the next time `Update` is called. Finally, the `Tick` method advances one frame; namely, it calls `Update` and `Draw`.

Properties

After covering all of the methods, properties are naturally the next item on the list. You've already seen the `Services` property, which enables you to add new services to the game and query for existing ones. You've also already seen the `Content` and `GraphicsDevice` properties, which store the default content manager and graphics device.

A property called `InactiveSleepTime` gives you some control of how your game handles itself when it is not the foreground window. This value tells the system how long to "sleep" when it is not the active process before checking to see if it is active again. The default value of this is 20 milliseconds. This is important on Windows were you can have many processes run at once. You don't want your game to run at full speed when it isn't even active.

Speaking of being active, the `IsActive` property tells you the current state of the game. This maps to the `Activated` and `Deactivated` events, too, as it turns true during `Activated` and false during `Deactivated`. Chapter 12, "Adding Interactivity with User Input" discusses the `IsMouseVisible` property, even though you can probably guess what it does.

The `LaunchParameters` property is used for Windows Phone 7 to get information about parameters required for launching, but this can be used for any platformand translates the command-line parameters on Windows into this object. It is a dictionary of key value pairs. On Windows, if your command line is as follows, the object would have a dictionary with three members:

```
game.exe /p /x:abc "/w:hello world"
```

The first member would have a key of "p" with a value of an empty string. The second member would have a key of "x" with a value of "abc." The third member has a key of "w" with a value of "hello world."

On Windows Phone 7, applications are launched with a URI that includes these parameters; for example, if your launch command is as follows, the object would have a dictionary with two members:

```
app://{appguid}/_default#/Main.xaml?myparam1=one&myparam2=two
```

The first member would have a key of "myparam1" and a value of "one." The second member would have a key of "myparam2" and a value of "two."

The last two properties are `IsFixedTimeStep` and `TargetElapsedTime`. Timing is so important to game development there is a whole section on that! Because anticipation is probably overwhelming, that section is next.

GameTime

You may not realize it, but a lot of things in a game depend on time. If you create a race game and your cars are going 60 miles per hour, you need to know how much to move them based on a given time. The framework tries to do a lot of the work of handling time for you.

There are two major ways to run a game, and in the framework, they are referred to as "fixed time step," and "variable time step." The two properties mentioned in the previous section—`IsFixedTimeStep` and `TargetElapsedTime`—control how time is handled. `IsFixedTimeStep` being true naturally puts the game into fixed time step mode, whereas false puts the game into variable time step mode. If you are in fixed time step mode, `TargetElapsedTime` is the target time for each frame. The defaults for projects are true for `IsFixedTimeStep` and 60 frames per second for `TargetElapsedTime` (which is measured in time, so approximately 16.6667 milliseconds).

What do these time steps actually mean? Variable time step means that the amount of time between frames is not constant. Your game gets one `Update`, then one `Draw`, and then it repeats until the game exits. If you noticed, the parameter to the `Update` method is the `GameTime`.

The `GameTime` object has three properties that you can use. First, it has the `ElapsedGameTime`, which is the amount of time that has passed since the last call to `Update`. It also includes `TotalGameTime`, which is the amount of time that has passed since the game has started. Finally, it includes `IsRunningSlowly`, which is only important in fixed time step mode.

During variable time step mode, the amount of time recorded in `ElapsedGameTime` passed to update can change depending on how long the frame actually takes (hence, the name "variable" time step).

Fixed time step is different. Every call to `Update` has the same elapsed time (hence, it is "fixed"). It is also different from variable time step in the potential order of `Update` and `Draw` calls. While in variable time step, you get one update for every draw call; in fixed time step, you potentially get numerous `Update` calls in between each `Draw`.

The logic used for fixed time step is as follows (assuming you've asked for a TargetElapsedTime of 60 frames per second).

`Update` is called as many times as necessary to catch up to the current time. For example, if your `TargetElapsedTime` is 16.667 milliseconds, and it has been 33 milliseconds since your last `Update` call, `Update` is called, and then immediately it is called a second time. `Draw` is then called. At the end of any `Draw`, if it is not time for an `Update` to occur, the framework waits until it is time for the next `Update` before continuing.

If at any time, the runtime detects things are going too slow (for example, you need to call Update multiple times to catch up), it sets the `IsRunningSlowly` property to true. This gives the game the opportunity to do things to run faster (such as rendering less or doing fewer calculations).

If the game gets extremely far behind, though, as would happen if you paused the debugger inside the `Update` call if your computer just isn't fast enough, or if your `Update` method takes longer than the `TargetElapsedTime`, the runtime eventually decides it cannot catch up. When this happens, it assumes it cannot catch up, resets the elapsed time, and starts executing as normal again. If you paused in the debugger, things should just start working normally. If your computer isn't good enough to run your game well, you should notice things running slowly instead.

You can also reset the elapsed time yourself if you know you are going to run a long operation, such as loading a level or what have you. At the end of any long operation such as this, you can call `ResetElapsedTime` on the game object to signify that this operation takes a while, don't try to catch up, and just start updating from now.

Notice that in Windows Phone 7 projects, the new project instead sets the `TargetElapsedTime` to 30 frames per second, rather than the 60 used on Windows and Xbox 360. This is done to save battery power, among other reasons. Running at half the speed can be a significant savings of battery life.

> **Note**
>
> Which time step mode you actually use is a matter of personal preference. We personally choose fixed time step mode, but either can work for any type of game. During performance measurement, though, you should use variable time step mode. This is discussed in Chapter 11, "Understanding Performance."

Game Loop

This chapter has been nothing but text so far. Words, words, and rambling—that just isn't exciting. Let's get something on the screen!

Update and Draw

The basic flow of your game is to initialize everything, and then call `Update` and `Draw` continually until you exit the game. This is what is called the game loop. You've already seen it in action in Chapter 2, "Sprites and 2D Graphics," without even realizing it most likely. To ensure you do realize it, let's do a slightly more complex example.

First, you need to add the XnaLogo.png file to your content project from the accompanying CD. Because you are drawing this image much like before, you need to declare a texture variable to hold it. This time, though, also declare a position variable, as follows:

```
Texture2D texture;
Vector2 position;
```

Of course, you need to load that texture, so in your `LoadContent` method add this line:

```
texture = Content.Load<Texture2D>("XnaLogo");
```

You should probably also initialize that position to something! Add the following to the `Initialize` method:

```
position = Vector2.Zero;
```

Finally, because you need to actually draw the image, replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(texture, position, Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

Running the project now shows you (as you might have guessed) the image in the upper left corner of the window. Because of the position variable in the `Draw` call, you are set up to get that thing moving! Add the following to the Update method:

```
position = new Vector2(position.X + (float)gameTime.ElapsedGameTime.TotalSeconds,
    position.Y + (float)gameTime.ElapsedGameTime.TotalSeconds);
```

Running the project now has the image slowly moving down and to the right, and if you leave it running long enough, it eventually goes off screen! If you want the image to bounce around the screen, though, it would be complicated code. Do you add to the position or subtract? Instead, store your direction in a new variable:

```
Vector2 velocity;
```

Then, update the Initialize method to include:

```
velocity = new Vector2(30, 30);
```

Finally, change your update method to:

```
position += (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
```

These updates provide faster movement and easier code, although the image still goes off screen if you let it. You need to detect if the image has gone off screen and make changes to ensure it "bounces" off the edges. This is the largest section of code you've seen so far, but it makes the image bounce around, so modify your `Update` call to this:

```
position += (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
if (!GraphicsDevice.Viewport.Bounds.Contains(new Rectangle(
    (int)position.X, (int)position.Y, texture.Width, texture.Height)))
{
    bool negateX = false;
    bool negateY = false;
    // Update velocity based on where you crossed the border
    if ((position.X < 0) || ((position.X + texture.Width) >
        GraphicsDevice.Viewport.Width))
    {
        negateX = true;
    }
    if ((position.Y < 0) || ((position.Y + texture.Height) >
        GraphicsDevice.Viewport.Height))
    {
        negateY = true;
    }
    // Move back to where you were before
    position -= (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
    if (negateX) velocity.X *= -1;
    if (negateY) velocity.Y *= -1;
    // Finally do the correct update
    position += (velocity * (float)gameTime.ElapsedGameTime.TotalSeconds);
}
```

Run the project now and you can see the image bounce around the screen! You simply check to see if the image is currently in the bounds of the viewport, and if it is not, check to see which edge it is currently over. Then, move it back to where it was before the update, swap the velocity axis of the sides you've crossed, and update the position again.

Components

As you can probably imagine, if you had to draw everything inside a single class, your code would quick become a mess of things to keep track of!

GameComponents

Luckily, the framework provides an easy way to encapsulate objects called game components. There are three types of game components you can use for your games: `GameComponent`, `DrawableGameComponent`, and `GamerServicesComponent`. The latter is discussed later, but let's dive into the others now.

First, you want to take the image-bouncing code you wrote and move it into a component, so in your main game project, right-click the project and select Add -> New Item. Choose Game Component from the list of templates (it might be easier to find if you choose the XNA Game Studio 4.0 section in the list on the left), name it BouncingImage.cs, and then click the Add button.

This adds a new file to your project with a new class deriving from `GameComponent`, which is close to what you want but not quite. Open up BouncingImage.cs (it should have opened when you added the component), and change it to derive from `DrawableGameComponent` instead:

```
public class BouncingImage : Microsoft.Xna.Framework.DrawableGameComponent
```

Now you can begin moving the code you used in your `Game` class to render your bouncing image to this component. Start by moving the three variables you added to the new `BouncingImage` class (texture, position, and velocity). Move the code initializing your two vectors into the new classes `Initialize` method and move the code were you modify the vectors in update to the new classes `Update` method. You need to do just a few things to complete your bouncing image component.

You need a way to load the texture, and `DrawableGameComponent` has the same virtual `LoadContent` method that the game has, so you can simply override it in your `BouncingImage` class now:

```
protected override void LoadContent()
{
    texture = Game.Content.Load<Texture2D>("XnaLogo");
    base.LoadContent();
}
```

Finally, all you need now is to draw the image. Just like `LoadContent`, `DrawableGameComponent` also has a `Draw` virtual method you can override:

```
public override void Draw(GameTime gameTime)
{
    spriteBatch.Begin();
    spriteBatch.Draw(texture, position, Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

As you might have seen already, this won't compile. The `spriteBatch` variable is declared in the game, and it is a private member variable. You can create a new sprite batch for this component, but it isn't necessary. If you remember back to earlier in this chapter, we talked about the `Services` property on the `Game` class.

Go back to the game1.cs code file and to the `LoadContent` method. Replace the loading of the texture (which you just did in the `BouncingImage` class) with the following line (directly after the sprite batch has been created):

```
Services.AddService(typeof(SpriteBatch), spriteBatch);
```

This adds a "service" of type `SpriteBatch` to the game class and uses the just created sprite batch as the provider of that service. This enables you to use the "service" from anything that has access to the game. Back in the `BouncingImage.Draw` method, before the `Begin` call, add the following:

```
SpriteBatch spriteBatch  = Game.Services.GetService(
    typeof(SpriteBatch)) as SpriteBatch;
```

Now that you have your drawing code in your component compiling, you can remove it from the `Draw` call in the game. It should have nothing but the `Clear` call and the `base.Draw` call now. With everything compiling, you can run your project and you will see absolutely nothing except a blank cornflower blue screen. This is because your component was never actually used! Back in the `Initialize` method in the game, add the following:

```
Components.Add(new BouncingImage(this));
```

That's it. Running the code gets you back to where you were before, but with everything much more encapsulated. It's also much easier to add new instances of the bouncing image; for example, add this to your games `Update` method:

```
if ( ((int)gameTime.TotalGameTime.TotalSeconds % 5) == 4)
    Components.Add(new BouncingImage(this));
```

Running it now adds a new set of bouncing images every 5 seconds (actually it adds quite a few because it adds one for every update that happens during that second). You can go ahead and remove that code; it is just an example of how easy it is to include more.

You might have noticed that you didn't actually have to do anything outside of add your component to the `Components` collection for it to start working magically. Your `Initialize` method is called for you, as is your `LoadContent` method, and the `Update` and `Draw` methods were called for each frame.

By default, components are updated and drawn in the order they were added, and they are always updated and drawn. However, these are all changeable, too. If you set the `DrawOrder` property, when the components are being drawn, the components with the lower `DrawOrder` values are drawn first. Similarly, if you use the `UpdateOrder` property, the components with the lower `UpdateOrder` value are updated first. The higher value these properties have, the later they happen in the list of components. If you want something drawn as the last possible component, you set the `DrawOrder` to `int.MaxValue,` for example. Of course, if you have more than one component with the same `UpdateOrder` or `DrawOrder`, they are called in the order they were added.

Of course, there might be times when you don't want your component to be drawn at all! If this is the case, you can simply set the `Visible` property to false, and your Draw override is no longer called until that property switches back to true. If you need to temporarily suspend updating for a while, you can just change the `Enabled` property to false!

There are also events (and virtual methods) to notify you when any of these properties change if you need to know about the changes.

> **Note**
>
> The `GameComponent` class has the same behavior as the `DrawableGameComponent` class without the extra methods and properties used for Drawing, such as `LoadContent` and `DrawOrder`.

Summary

Now you have a basic understanding of how game flow is executed and the order of operations of a normal game. You also understand components and how they interact. It is time to move into some 3D graphics before getting back to some of the more advanced 2D operations.

4

Introduction to 3D Graphics

Let's face it—graphics are cool. Graphics are a major reason why many developers become interested in gaming technology and creating games. The capability to create an image that can change over time is impressive. 3D graphics gives a false since of depth to an image on a flat 2D surface.

In this chapter, you learn:

- What makes up 3D graphics
- Basic mathematics used in 3D graphics
- Stages of the graphics pipeline
- Graphics profiles
- Using the GraphicsDevice and GraphicsAdapter objects
- Drawing each of the primitive types

3D Graphics in XNA Game Studio

Developing 3D graphics is similar to creating real-time art. Even more impressive are the detailed and interactive 3D graphics found in modern real-time computer games. It is impressive when you compare it against computer graphics of ten years ago. Console and PC games were limited to a fixed set of hardware features, and resolutions were quite low and far from today's high resolution, multisampled, antialiased, and almost realistic images.

With all of the new fancy hardware-dependent graphical improvements of the past decade also came an increase in complexity when creating graphics for games. Developers deal with many complexities, even when drawing simple graphics on the screen.

XNA Game Studio has attempted to limit these complexities in some cases and in others to solve them all together. As discussed in previous chapters, the capability to have a basic graphics window display a clear color is trivial and requires only that you create a new project for one of the supported platforms. To get to that point, using C++ with DirectX or OpenGL can take hundreds of lines of code. The graphics APIs exposed by XNA Game Studio are easy to use and often lead to the developer to fall into the correct solution using them.

> **Note**
>
> If you have used XNA Game Studio before, you will notice a number of API differences in XNA Game Studio 4.0. A tremendous amount of work and energy went into updating the graphics APIs to be even easier to use and to support the new Windows Phone 7 platform.

In this chapter, we cover the basics developers should know before diving into creating 3D graphics. We start with defining 3D graphics and their makeup. Then, we focus on some basic math that is required when working on 3D graphics. Next, we cover the different stages that are part of the graphics pipeline. We finish by covering the XNA Game Studio `GraphicsAdapter` and `GraphicsDevice` classes and drawing the first 3D primitives on the screen.

What Are 3D Graphics?

The answer to this question might seem simple, but it might not be obvious to those who have no experience creating 3D games or applications. In reality, 3D graphics are just an illusion. They are flat 2D images on a computer monitor, television screen, or phone display.

In the real world, your eyes work together to create an image that contains visual perspective. Objects appear to be smaller the farther they are away from you physically. Imagine you are standing in the middle of perfectly straight and flat railroad tracks. If you look down at your feet, the rails appear to be some distance apart. As you let your eyes move towards the horizon and look farther down the tracks, it appears that the track rails become closer to each other the farther they are away from you. Having two eyes adds to this depth perception because each eye has a slightly different view. They help to see slightly different angles of objects. This difference is even greater for objects near to you. If you alternate closing each eye, you see objects move left and right.

When you create 3D graphics, perspective is an illusion. When you are working with 3D graphics concepts, remember that you are really creating a 2D image that is displayed on the screen. Many parts of the graphics pipeline have nothing to do with 3D at all and work only on the pixel level. A friend of mine once described to me a helpful way to look at developing 3D graphics games. He related creating 3D graphics to how great painters create stunningly realistic pieces of art. Although to a viewer standing still the painting can appear to represent real 3D objects in the world, it is still just layers of paint on flat canvas. They are illusions just like computer-generated 3D graphics.

So what are 3D graphics? They are computer-generated images that give the appearance of depth and perspective, but in reality, they are just flat 2D images when displayed.

Makeup of a 3D Image

In a Chapter 2, “Sprites and 2D Graphics”, we described how sprite textures are used to draw 2D images on the screen. You can think of this concept as placing stickers on the screen and moving them around.

3D graphics images are generated differently. At their core, they are a bunch of triangles drawn on the screen. If this sounds simplistic, it is in order to be fast. Modern graphics hardware can be so fast because the types of operations it performs are limited when compared to general-purpose processors used for CPUs in PCs. Graphics hardware has dedicated hardware for these operations and are designed to process millions of triangle and pixel operations.

Groups of triangles drawn on the screen are commonly called geometry. These collections of geometry can represent almost anything from a box, to a space ship, to a human marine storming the beaches of Normandy.

Each triangle is made up of three vertices. Each vertex represents one of the corners of the triangle and must contain at least the position of itself. A vertex can also contain other data such as color, texture coordinates, normals, and so on.

Although triangles determine the shape of what is displayed on the screen, there are many factors that determine the final color of pixels the triangle covers on the screen. Some of those factors such as texturing, lighting, and blending are covered in Chapter 6, "Built-In Shader Effects" and Chapter 7, "States, Blending, and Textures."

> **2D Sprites Are Actually in 3D!**
>
> When you draw 2D sprites in XNA Game Studio, you use the 3D graphics pipeline. In the past, 2D graphics were drawn using a technique called blitting, which was copying memory from a surface such as a texture to the display memory. With today's graphics cards, it is easier and faster to just draw two-textured triangles when you want to draw a texture on the screen. Because sprites are also 3D geometry, many of the topics in this chapter are also applicable to them.

3D Math Basics

Those who are new to 3D graphics might ask the question, "Do I need to know math to create 3D graphics?" The simple answer is "Yes, there is a level of mathematics that is required when you are working on a 3D game." Can you get by without knowing much? Absolutely, and we have seen many examples where people have created a 3D game but don't necessarily understand how they are getting the results they see. This often leads to questions such as, "Why is my model orbiting around the camera when I wanted it to rotate on its axis like a planet?"

Without a doubt, having a better understanding of underlying mathematics leads you to be less confused, more quickly understand new concepts, and be more productive.

This section attempts to strike a balance of theory and practical use. The goal is to give you the higher level concepts that are used throughout 3D graphics without having to explain the details of how to do specific mathematical operations such as vector arithmetic or matrix multiplication. There are countless educational sources on these operations, and they are already implemented in the math types provided in XNA Game Studio. We spend our time focusing on what these operations mean geometrically and how they are used within 3D graphics.

Coordinate Systems

We already used one type of coordinate system, which were screen coordinates when drawing sprites on the screen. In screen coordinate space, there are two dimensions: one for each of the X and Y directions. In screen coordinates, the X direction increases in value from the left side of the screen to the right, and the Y increases in value from the top of the screen to the bottom.

There are two main types of 3D coordinate systems used in computer graphics. These two different types differ in direction from the positive Z points relative to the X and Y axes. The two types of coordinate systems are called right-handed and left-handed (see Figure 4.1). Both of these systems contain three directions for the X, Y, and Z axes. The three axes converge at a central point called the origin where their values equal 0. The values along each axis either gain or lower in value at regular intervals depending on whether you are moving in the positive or negative direction along that axis.

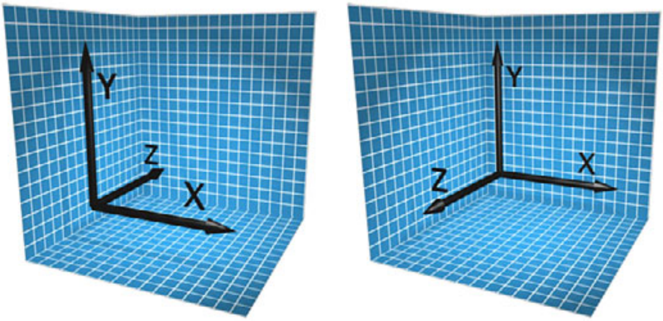


Figure 4.1 Left- and right-handed coordinate systems

To visualize these two different coordinate systems, let's do a quick exercise. Take your right hand and make a fist. Turn your hand so your bent fingers face you and the top of your hand faces away from you. Now, extend your thumb. It should point directly to your right. Your thumb represents the positive X axis. Now extend your index finger, also known as your pointer finger, directly up in the air. This represents the positive Y direction. Your hand should look like it is making a capital L now with your thumb facing right and representing the positive X axis and your index finger pointing upwards representing the positive Y axis. Finally, extend your middle finger and point it directly at yourself. Your middle finger represents the positive Z axis (see Figure 4.2).

Imagine lines that extend from the tips of all three of your fingers; they should be 90 degrees apart from each other forming right angles between themselves. Having each axis be at right angles from each other creates what is called an orthogonal coordinate system. In this case, you created a right-handed coordinate system. This is the coordinate system used by default in XNA Game Studio. In a right-handed coordinate system, the negative Z axis is used for the forward direction.

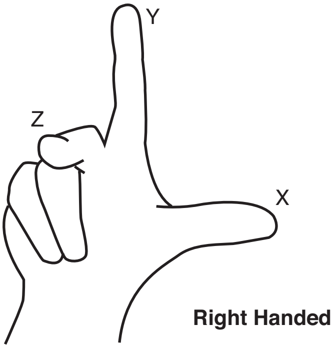


Figure 4.2 Right fist forming the right-handed coordinate system

> **Note**
>
> DirectX by default uses a left-handed coordinate system. The properties and methods in XNA Game Studio assume you are using a right-handed coordinate system. Some methods also provide a left-handed equivalent for those who want to use a left-handed system.

To visualize a left-handed system, follow the previous instruction except face your palm away from yourself but still make a capital L with your thumb and index finger for the positive X and Y axes. Now when you extend your middle finger to form the positive Z axis, notice that it now points away from you. In a left-handed coordinate system, the positive Z axis is used for the forward direction (see Figure 4.3).

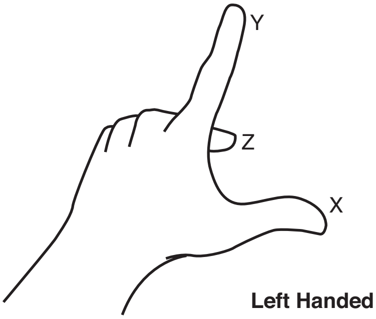


Figure 4.3 Left fist forming the left-handed coordinate system

> **Note**
>
> Although we just stated that Z axis is used for the forward direction, this is not always true. In some 3D art creation packages, the Z axis actually represents the up direction. The good news is that this difference is accounted for when XNA Game Studio loads models from those specific 3D content packages.

Vectors in 3D Graphics

A vector is a mathematical geometric construct that consists of multidimensional values that provide a magnitude and direction. That sounded a little too much like a math class. You can think of vectors as a set of floating point values used to represent a point or direction in space. Use groups of vectors to represent the triangles that make up the geometry in your game.

You have already used vectors in Chapter 2, "Sprites and 2D Graphics." The `Vector2` type, which contains two float values for `X` and `Y` is used to express the position to draw the sprites. XNA Game Studio also provides `Vector3` and `Vector4` types, which contain three and four components each.

XNA Game Studio uses free vectors. Free vectors are represented by a single set of components equal to the number of dimensions of the vector. This single set of components is able to express a direction and a magnitude. By contrast, in mathematics, another type of vector exists called a bounded vector, which is represented by two sets of components for both the start point and the end point. Because they are not often used in computer graphics, we focus on free vectors, which we call just "vectors."

> **What Units Are Vectors In?**
>
> Vectors are in whatever unit you want them to be in. The important rule is to be consistent throughout your game and even your art pipeline. If one artist creates buildings for your game and one unit equals a meter, and another artist creates your game characters with one unit equaling one inch, the buildings are going to look small or your characters are going to look big. If you are working on a team, it is helpful to decide on what one unit is equal to in your game.
>
> If you are not able to author or export your art content yourself, the content pipeline scales your models for you given a scaling factor.

Vector4 the Four Dimensional Vector

As we mentioned, XNA Game Studio supports three types of vectors: `Vector2`, `Vector3`, and `Vector4`. `Vector2` has two dimensions, so use it in 2D graphics. `Vector3` has three dimensions, and is used it in 3D graphics. What should `Vector4` be used in?

A `Vector4` like the `Vector3` type contains the `X`, `Y`, and `Z` values. The fourth component, called the homogeneous component and represented by the `W` property, is not used for space time manipulation unfortunately. The fourth component is required when multiplying the vector by a matrix, which has four rows of values. Matrices and vector multiplication are discussed later in this chapter.

Point Versus Direction and Magnitude

When working with `Vector3` and `Vector4` values, they can represent a 3D point in space or a direction with a magnitude. When you use a vector as the vertex position of a triangle, it represents a point in space. When a vector is used to store the velocity of an object, it represents a direction and a magnitude. The magnitude of a vector is also commonly referred to as the length of the vector and can be accessed using the `Length` property of all of the vector types.

It is often useful to have a vector that represents a direction and magnitude to have a length of 1. This type of vector is called a unit vector. Unit vectors have nice mathematical properties in that they simplify many of the equations used with vectors so that their operations can be faster, which is important when creating real-time graphics in games. When you change a vector, so its direction stays the same but the length of the vector is set to one, it is called normalizing the vector. A `Normalize` method is provided for each of the vector types.

Vector Addition

When you add two vectors A and B together, you obtain a third vector C, which contains the addition of each of the vector components. Vector addition, like normal algebraic addition, is commutative meaning that A + B is equal to B + A.

Geometrically vector addition is described as moving the tail of the B vector, which is at the origin, to the head of the A vector, which is the value the A vector represents in space. The resulting vector C is the vector from the tail of A, which is the origin, to the head of B.

In Figure 4.4, vector A contains a value of { 2, 1 } and vector B contains a value of { 1, 3 }. The resulting vector C contains the value { 3, 4 }.

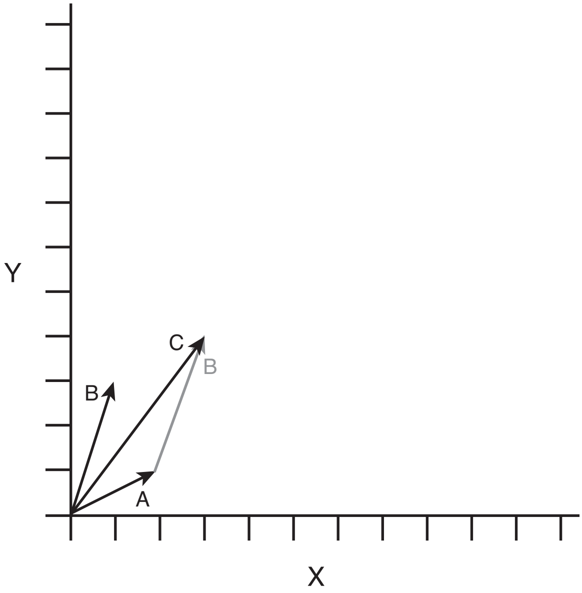


Figure 4.4 Example of vector addition

If any of the components of the vector are negative, the values are added together in the same way. The resulting value can have negative components.

Vector Subtraction

When you subtract two vectors A and B from each other, you obtain a third vector C, which is the difference of each of the vector components. Vector subtraction is not commutative meaning that A – B is not equal to B – A.

Geometrically vector subtraction is described as moving the head of vector B to the head of the A vector. The resulting vector C is formed from the tail of A, which is at the origin, to the tail of B.

In Figure 4.5, vector A contains a value of { 3, 4 } and vector B contains a value of { 1, 3 }. The resulting vector C contains the value { 2, 1 }.

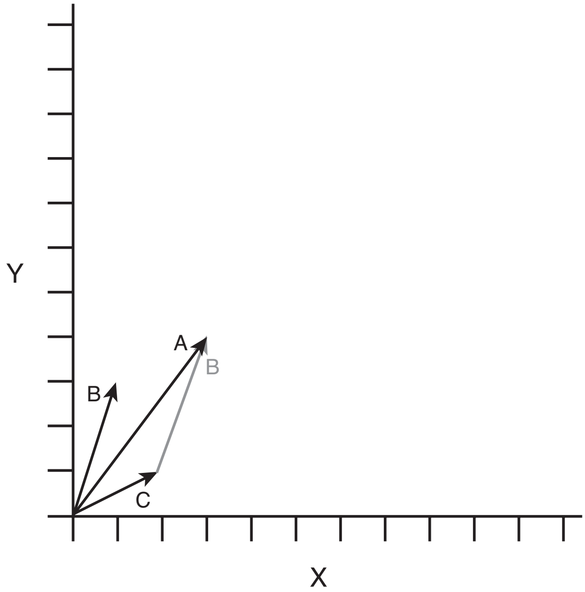


Figure 4.5 Example of vector subtraction

Vector Scalar Multiplication

Vectors can be multiplied by a single number called a scalar. This scalar value is multiplied against each component to produce a new vector with the same direction but whose length has been multiplied by the scalar.

Geometrically the multiplication of a vector by a scalar can be described as taking the original vector and stretching or shrinking the length by the scalar value.

In Figure 4.6, vector A has a value of { 1, 2 }. When multiplied by the scalar of 2, the new vector S has a value of { 2, 4 }.

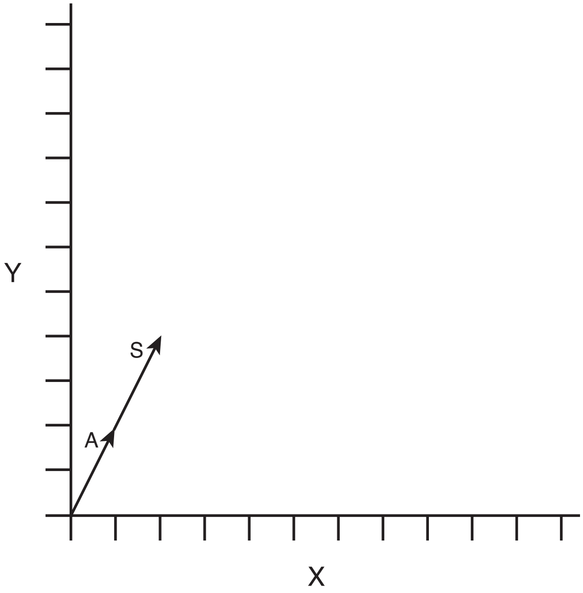


Figure 4.6 Example of vector scalar multiplication

Vector Negation

To negate a vector, each of the component's signs are changed to the opposite. Positive components become negative and negative components become positive. This is the same as multiplying the vector by a value of negative one. The negation of a vector is often written –A, where A is the vector name. Each of the vector types contains a method called `Negate`, which returns the negation of the given vector.

Geometrically the negation of a vector represents a vector in the opposite direction with the same length.

In Figure 4.7, vector A has a value of { 2, -1 }. When negated, it produces vector N, which has the value { -2, 1 }.

Vector Dot Product

The dot product, also called the scalar product, creates a scalar value from multiplying each of the vector components with each other and adding the results together. The dot product is written in the form A • B, pronounced as A dot B. The resulting scalar value is equal to the cosine of the angle times the length of A times the length of B. As we mentioned before, using normalized vectors simplifies some linear algebra equations, and this is one of them. To determine the angle between two vectors, first normalize each of the vectors.

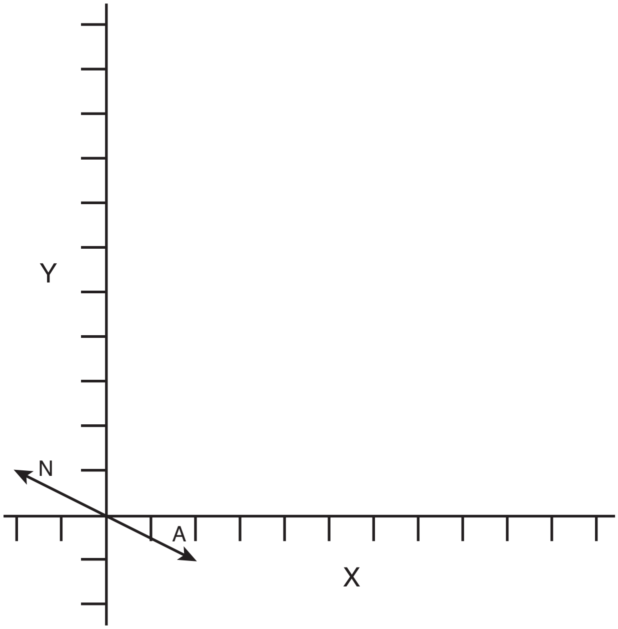


Figure 4.7 Example of vector negation

Then, take the dot product to return the cosine of the angle. Use the arccosine to find the angle value in radians.

Geometrically you can think of the dot product of two vectors A and B as the length of vector A in the direction of vector B. Because the vector dot product is commutative, it has the same value as the length of vector B in the direction of vector A. This of this as one vector casting a shadow onto the other vector.

In Figure 4.8, the dot product A • B is both a scalar value related to the angle between the vectors as well as the length of vector A in the direction of vector B.

Vector Cross Product

The vector cross product of vectors A and B produce a resulting vector C that is perpendicular to the plane that both A and B are located on. The cross product is commonly written as A × B pronounced A cross B. The cross product is helpful when building orthonormalized coordinate spaces. As we discussed, 3D coordinate spaces contain three unit vectors for the X, Y, and Z axis. Each of these unit vectors can be created by taking the cross product of the other two axes. The Z axis is created from the result of X cross Y. Y from Z cross X. X from Y cross Z. The cross product is not commutative nor is it associative, so A × B is not equal to B × A. Taking the cross product of B × A produces a perpendicular vector that is the negation of A × B.

In Figure 4.9, notice that vectors A × B produces a perpendicular vector C while B × A produces a vector equal to –C.

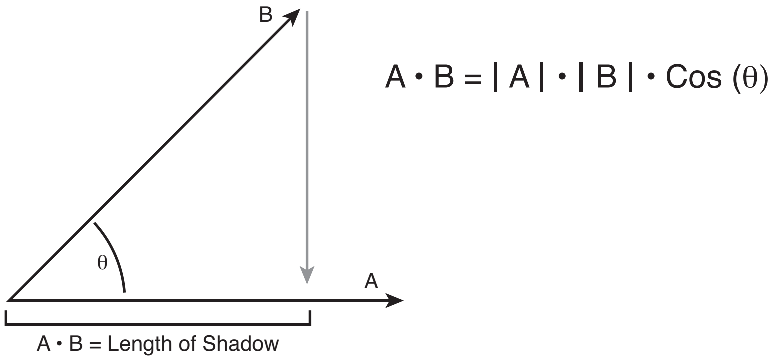


Figure 4.8 Example of vector dot product

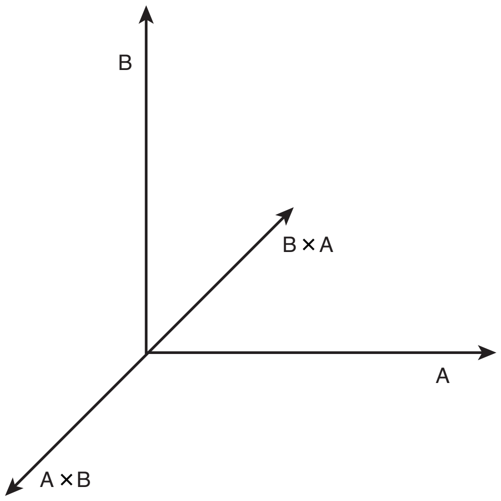


Figure 4.9 Example of vector cross product

Vectors in XNA Game Studio

The vector types in XNA Game Studio provide a number of helpful properties and methods to perform the calculations we have been discussing and more that we cover throughout this book. Tables 4.1, 4.2, and 4.3 contain the fields, properties, and methods of the `Vector3` type. The `Vector2` and `Vector4` types have similar fields, properties, and methods.

Table 4.1 **Fields of Vector3**

Field	Type	Description
`X`	`float`	The `X` component of the vector
`Y`	`float`	The `Y` component of the vector
`Z`	`float`	The `Z` component of the vector

Table 4.2 **Properties of Vector3**

Property	Type	Description
`UnitX`	`Vector3`	Static property that returns a vector with the values { 1, 0, 0 }
`UnitY`	`Vector3`	Static property that returns a vector with the values { 0, 1, 0 }
`UnitZ`	`Vector3`	Static property that returns a vector with the values { 0, 0, 1 }
`Right`	`Vector3`	Static property that returns a unit vector pointing in the right direction { 1, 0, 0 }
`Left`	`Vector3`	Static property that returns a unit vector pointing in the left direction { –1, 0, 0 }
`Up`	`Vector3`	Static property that returns a unit vector pointing in the up direction { 0, 1, 0 }
`Down`	`Vector3`	Static property that returns a unit vector pointing in the down direction { 0, –1, 0 }
`Forward`	`Vector3`	Static property that returns a unit vector pointing in the forward direction { 0, 0, –1 }
`Backward`	`Vector3`	Static property that returns a unit vector pointing in the backward direction { 0, 0, 1 }
`One`	`Vector3`	Static property that returns a vector with the values { 1, 1, 1 }
`Zero`	`Vector3`	Static property that returns a vector with the values { 0, 0, 0 }

Table 4.3 **Abbreviated Table of Vector3 Methods**

Method	Description
`Add`	Returns the additions of two vectors.
`Subtract`	Returns the subtraction of two vectors.
`Negate`	Returns the negation of two vectors.
`Dot`	Returns the scalar result from the dot product of two vectors.
`Cross`	Returns a perpendicular vector that is the cross product of two vectors.

Table 4.3 **Abbreviated Table of Vector3 Methods**

Method	Description
`Multiply`	Multiplies each component of a vector by a scalar value or another vector.
`Divide`	Divides each component of a vector by a scalar value or another vector.
`Normalize`	Normalizes a vector to have a unit length of one with the same direction as the starting vector.
`Length`	Returns the length of the vector.
`LengthSquared`	Returns the length of the vector squared. It is a less expensive operation to calculate the length squared of a vector. Use for relative comparisons.
`Distance`	Returns the distance between two vectors.
`DistanceSquared`	Returns the distance between two vectors squared. It is a less expensive operation to calculate the distance squared. Use for relative comparisons.
`Min`	Returns a vector whose components are the smallest from each of the given vectors.
`Max`	Returns a vector whose components are the largest from each of the given vectors.
`Transform`	Transforms a vector that represents a point by a matrix.
`TransfromNormal`	Transforms a vector that represents a normal by a matrix.

> **Note**
>
> Many of the math methods provided in XNA Game Studio provide overloads, which take the parameters as references and return the result as an out parameter. Using these versions of the methods can save the overhead of many copies of the structures passed to the method. Use these versions in your time-critical sections of your game where you have performance problems.

Matrix

In mathematics, a matrix is rectangle group of numbers called elements. The size of the matrix is expressed in the number of rows by the number of columns. In 3D graphics, the most common type of matrix is the 4 by 4 matrix, which contains 16 `float` values. The XNA Game Studio Matrix structure is a 4 by 4 matrix. A matrix has a number of mathematical applications in a number of fields including calculus and optics. For our purposes, we focus on the use of a matrix in linear algebra because of how useful this becomes in computer graphics.

In computer graphics, the matrix is used to represent linear transformations, which are used to transform vectors. These linear transformations include translation, rotation, scaling, shearing, and projection. The elements of the matrix create their own coordinate spaces by defining the direction of the X, Y, and Z directions along with the translation from the origin. Using a matrix that is 4 by 4 in size enables all of the linear transforms to be combined into a single matrix that is used to transform position and direction vectors.

In XNA Game Studio, the matrix structure is row major meaning that the vectors that make up the X, Y, and Z directions and the translation vector are laid out in each row of the matrix. Each row represents a different direction in the coordinate space defined by the matrix.

In the first row, the X vector represents the right vector of the coordinate space. In the second row, the Y vector represents the up vector of the coordinate space. In the third row, the Z vector represents the backward vector of the coordinate space. The forward vector is actually the negation of the Z vector because in a right-handed coordinate space, the Z direction points backwards. The forth row contains the vector to use for the translation of the position. Figure 4.10 shows the vector components of a matrix.

$$\begin{array}{r|cccc|}
\text{Right} & R_X & R_Y & R_Z & R_W \\
\text{Up} & U_X & U_Y & U_Z & U_W \\
-\text{Forward} & -F_X & -F_Y & -F_Z & -F_W \\
\text{Translation} & T_X & T_Y & T_Z & T_W
\end{array}$$

Figure 4.10 Vector components of a matrix

Many of the transforms produce what is called an orthogonal matrix where the X, Y, and Z directions are all 90 degrees from each other. In addition there are some transforms, which contain both orthogonal and normalized direction vectors producing an orthonormalized matrix.

Identity

The identity matrix, also called the unit matrix, contains elements with the value of one from the top left diagonal down to the bottom right. The rest of the elements in the matrix are all zeros (see Figure 4.11).

$$\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

Figure 4.11 Identity matrix

When the identity matrix is multiplied by any other matrix, the result is always the original matrix (see Figure 4.12).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 2 & 3 & 1 & 2 \\ 4 & 5 & 1 & 7 \\ 2 & 9 & 3 & 6 \\ 8 & 7 & 5 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 4 & 5 & 1 & 7 \\ 2 & 9 & 3 & 6 \\ 8 & 7 & 5 & 1 \end{bmatrix}$$

Figure 4.12 Identity matrix multiply

The identity matrix is an orthonormalized matrix that defines the unit directions for X, Y, and Z for the unit coordinate space (see Figure 4.13).

$$\begin{array}{r} \text{Right} \\ \text{Up} \\ -\text{Forward} \\ \text{Translation} \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{array}{ll} \{1, 0, 0, 0\} & \text{Direction} \\ \{0, 1, 0, 0\} & \text{Direction} \\ \{0, 0, 1, 0\} & \text{Direction} \\ \{0, 0, 0, 1\} & \text{Position} \end{array}$$

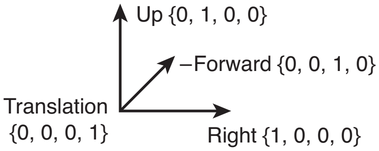


Figure 4.13 Right-handed unit coordinate space

The identity matrix is a base starting point for many of the other types of transforms.

Translation

A translation matrix is a type of matrix that is used to translate or move a vector from one location to another. For example if a vector contains the values { 1, 2, 3 }, a translation of { 2, 1, 0 } moves the vector to { 3, 3, 3 }.

In a translation matrix, the last row contains the values to translate by (see Figure 4.14).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_X & T_Y & T_Z & 1 \end{bmatrix}$$

Figure 4.14 Translation matrix

When a vector is multiplied by a translation matrix, the result is a vector with a value equaling the original vector's value plus the translation of the matrix.

Rotation

A rotation matrix transforms a vector by rotating it. Rotation matrices come in the form of rotations around the X, Y, or Z axes along with rotation around an arbitrary axis. Each type of rotation matrix has its own element layout for defining the rotation. Figures 4.15, 4.16, and 4.17 show the different types of axis rotation matrices.

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \text{Cos}\ominus & \text{-Sin}\ominus & 0 \\ 0 & \text{Sin}\ominus & \text{Cos}\ominus & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.15 X axis rotation matrix

$$R_Y = \begin{bmatrix} \text{Cos}\ominus & 0 & \text{Sin}\ominus & 0 \\ 0 & 1 & 0 & 0 \\ \text{-Sin}\ominus & 0 & \text{Cos}\ominus & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.16 Y axis rotation matrix

$$R_Z = \begin{bmatrix} \text{Cos}\ominus & \text{-Sin}\ominus & 0 & 0 \\ \text{Sin}\ominus & \text{Cos}\ominus & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.17 Z axis rotation matrix

When a vector is multiplied by a rotation matrix, the resulting vector's value is equal to the original vector value rotated around the defined axis.

In XNA Game Studio, the creation of rotation matrices is as simple as calling one of the static methods provided by the `Matrix` structure, such as `CreateRotationX`. The specified angles are in radian units. In radians, 2 Pi units is equal to 360 degrees. Pi is a mathematical constant, which is the ratio of a circle's circumference to the diameter of the circle. The value of Pi is around 3.14159. To convert between radians and degrees, use the `MathHelper.ToRadians` and `MathHelper.ToDegrees` methods.

Rotations are sometimes referred to in terms of yaw, pitch, and roll. These represent rotations around the current coordinate spaces right, up, and forward vectors, which are not necessarily the same as the unit X, Y, and Z axes. For example, if an object is already rotated 45 degrees around the Y axis, the forward vector is not in the negative Z direction anymore. It is now halfway between negative Z and negative X (see Figure 4.18).

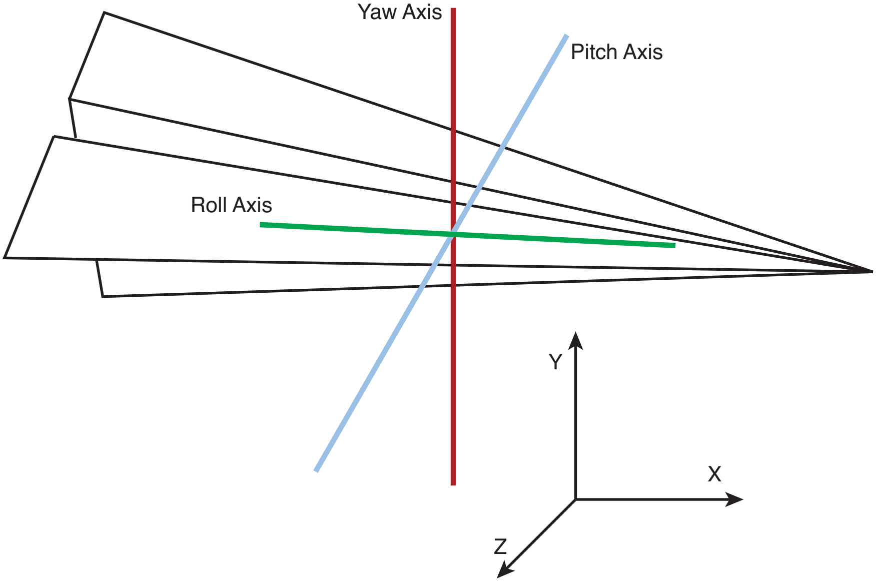


Figure 4.18 Yaw, pitch, and roll vectors for an oriented object

Rotating this object around the X axis is not the same as rotating around the pitch vector, because the pitch vector no longer is equal to the unit X vector.

Scale

A scale matrix transforms a vector by scaling the components of the vector. Like the identity matrix, the scale matrix uses only the elements in the diagonal direction from top left to lower right. The rest of the elements are all zeros (see Figure 4.19).

$$\begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.19 Scale matrix

When a vector is multiplied by a scale matrix, each component in the vector is scaled by a corresponding element in the matrix. A scale matrix does not have to be uniform in all directions. Nonuniform scale is also possible where some axis directions are scaled more or less than others.

Combining Matrix Transforms

Matrices cannot only be multiplied with vectors to transform them, but can also be combined together to form a new linear transform. The resulting matrix can then be used to transform the vector. The combined transforms on the vector have the same effect as multiplying each against the vector in succession.

Multiple matrices are combined to create a complex transform that is finally multiplied with the vector. This is beneficial when you are transforming thousands of vectors that represent the vertices of triangles in geometry. The resulting combined matrix needs to be calculated only once and can be used for all of the geometry.

When multiplying two matrices A by B, the number of columns in the first matrix A has to equal the number of rows in matrix B. In XNA Game Studio, this is not a concern because the `Matrix` structure is 4 by 4 square.

> **Note**
>
> When multiplying a vector with a matrix, the vector is treated as a 1 by 4 matrix.

Matrix multiplication is not commutative meaning that A × B is not the same as B × A, which makes sense if you think about it geometrically. If you have a rotation matrix R and multiply it by a translation matrix T, the resulting matrix first rotates an object and then translates the object. If you reverse the order, the object first translates and then rotates causing the object to orbit. The order you combine matrices is important. In general, you want to first scale, then rotate, and finally translate (see Figure 4.20).

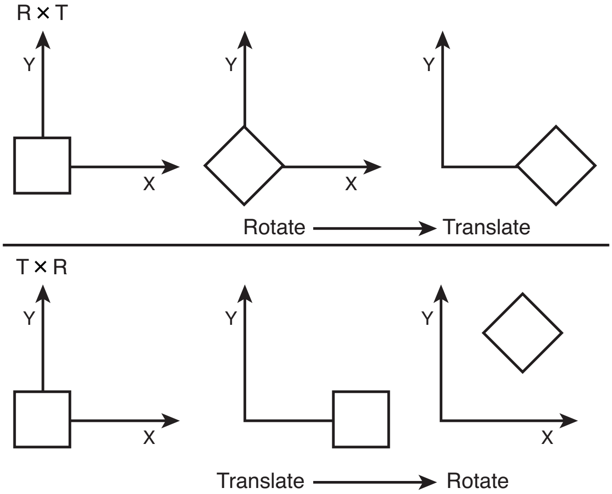


Figure 4.20 Results of multiplying by rotation and translation matrices in both orders

Manipulating Vectors with Matrices

Matrices are useful in transforming vectors. As we have discussed, vectors can be used for position data for things like the position of triangles that make up geometry in the game or they can be used for directions for things like the direction and velocity a car is moving. Transforming these two types of vectors occurs slightly differently and has to do with the fourth component of the vector, the homogeneous W component.

As we discussed, 4 by 4 matrices are used because they have the capability to hold multiple types of linear transforms in a single matrix. To be able to multiply a vector by a matrix, the number of components of the vector must equal the number of rows of the matrix. The fourth row of the matrix holds the translation of the matrix.

For vectors that represent position, it is important that the translation of the matrix affects the vector that is multiplied. Because of this, the `W` component of vectors that represent positions should have a value of 1. This enables the matrix multiply to include the affect of the translation elements of the matrix.

For vectors that represent a direction and magnitude, they should not be translated when multiplied by a matrix. Because of this, the `W` component of the vectors that represent direction vectors should have a value of 0. This enables the matrix multiply to not allow the translation of the matrix to affect the vector.

Often in your game, you will not want to store your position and direction vectors as `Vector4` types. Storing these values as a `Vector3` saves you the memory of an extra `float` component. It also saves you the trouble of having to set the `W` component. To determine whether the vector is used for position or as a direction, the `Vector3` structure provides two transform methods. The first called `Transfrom` takes a matrix and expects the input value to be a position vector. It expands the `Vector3` into a `Vector4`, sets the `W` component to 1, and then multiplies the input vector by the matrix returning the resulting vector as a `Vector3` after dropping the added `W` component. The other method `TransformNormal` works in the same way except it expects a direction vector and sets the `W` component to 0. Tables 4.4, 4.5, and 4.6 contain the fields, properties, and methods of the `Matrix` type.

Table 4.4 **Fields of Matrix**

Field	Type	Description
`M11`	`float`	Element in the first row and first column
`M12`	`float`	Element in the first row and second column
`M13`	`float`	Element in the first row and third column
`M14`	`float`	Element in the first row and fourth column
`M21`	`float`	Element in the second row and first column
`M22`	`float`	Element in the second row and second column
`M23`	`float`	Element in the second row and third column
`M24`	`float`	Element in the second row and fourth column

Table 4.4 **Fields of Matrix Continued**

Field	Type	Description
`M31`	`float`	Element in the third row and first column
`M32`	`float`	Element in the third row and second column
`M33`	`float`	Element in the third row and third column
`M34`	`float`	Element in the third row and fourth column
`M41`	`float`	Element in the fourth row and first column
`M42`	`float`	Element in the fourth row and second column
`M43`	`float`	Element in the fourth row and third column
`M44`	`float`	Element in the fourth row and fourth column

Table 4.5 **Properties of Matrix**

Property	Type	Description
`Identity`	`Matrix`	Returns an identity matrix
`Right`	`Vector3`	Returns the right vector of the matrix
`Left`	`Vector3`	Returns the left vector of the matrix
`Up`	`Vector3`	Returns the up vector of the matrix
`Down`	`Vector3`	Returns the down vector of the matrix
`Forward`	`Vector3`	Returns the forward vector of the matrix
`Backward`	`Vector3`	Returns the backward vector of the matrix
`Translation`	`Vector3`	Returns the translation vector of the matrix

Table 4.6 **Abbreviated Table of Matrix Methods**

Method	Description
`CreateTranslation`	Returns a translation matrix from a given translation vector
`CreateRotationX`	Returns a rotation matrix around the X axis in the given amount
`CreateRotationY`	Returns a rotation matrix around the Y axis in the given amount
`CreateRotationZ`	Returns a rotation matrix around the Z axis in the given amount
`CreateFromAxisAngle`	Returns a rotation matrix around the given axis in the given amount
`CreateFromYawPitchRoll`	Returns a rotation matrix from the given yaw, pitch, and roll amounts

Table 4.6 **Abbreviated Table of Matrix Methods** **Continued**

Method	**Description**
`CreateScale`	Returns a rotation matrix from the given scale
`Multiply`	Returns the result of multiplying two matrices together

Graphics Pipeline

The graphics pipeline is the set of operations that occur from when you request to draw some geometry that turns the triangles into pixels drawn on the screen or other render target. It is important to understand what is occurring in the pipeline so you can achieve the graphical results you desire. We briefly cover the important stages of the pipeline to give you a good idea about how triangles turn into pixels (see Figure 4.21). As we progress, some of the stages are covered in more detail.

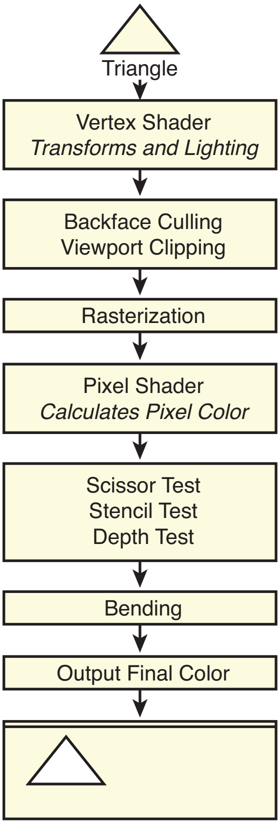

Triangle
Vertex Shader
Transforms and Lighting
Backface Culling
Viewport Clipping
Rasterization
Pixel Shader
Calculates Pixel Color
Scissor Test
Stencil Test
Depth Test
Bending
Output Final Color

Figure 4.21 High-level graphics pipeline from triangle to pixels on the screen

Graphics Card

With today's technology, almost all the graphics' pipeline computations occur in a specialized piece of hardware called a graphics card. Graphics cards contain a specialized processor called the graphics processing unit (GPU).

Graphics cards became popular towards the end of the 1990s when 3D computer games became more popular. Today, most PC computers contain some type of GPU. Sometimes the GPU is integrated onto the main motherboard of the machine.

There are several types of graphics cards on the market today. Having so many different graphics cards to support becomes a problem when trying to write games and other applications that want to utilize the hardware. 3D graphics APIs, like those exposed by the XNA Game Studio, enable developers to program their code once and have that code run in a similar manner across many different graphics cards.

Although the XNA Game Studio provides a single API set to communicate with several graphics cards, the behavior among these cards can vary greatly from card to card. To solve this problem, XNA Game Studio 4.0 introduced the concept of graphics profiles. Graphics profiles are discussed later in this chapter.

Vertex Shader

After you request to draw the triangles that make up an object, such as a teapot, each of these triangles are sent individually to the vertex shader. The vertex shader is a small program run on the graphics hardware that is responsible for transforming the triangle positions from the 3D model space they were defined in into a projected screen space so they can be drawn as pixels. The vertex shader is also responsible for calculating lighting values for the triangle that are used when determining the final color to output pixel color.

The vertex shader is programmable and controlled by creating a small vertex program that is compiled and set to the graphics hardware. Creating your own custom vertex shaders are covered in Chapter 8, "Introduction to Custom Effects".

World Transform

World space defines where the geometry is translated, how it is rotated, and how much it is scaled. All of these are linear transforms from their defined positions in model space.

To move from model space to world space, a world matrix is used to transform the vectors that make up the triangles in the geometry. Each set of geometry that make up objects in your game, like the teapot, generally have their own world matrix that translates, rotates, and scales the object.

View Transform

The geometry is now positioned in the world, but where the viewer is located? View space determines where the 3D scene is viewed from. Often it is helpful to think of view space as the view from a camera in the scene.

To move from world space to view space, a view matrix is used to transform the geometry. There are different types of view matrices that cause the scene to be viewed in different ways. Typically your scene utilizes one view matrix for all of your geometry. For more information on the view matrix, see Chapter 5, "Lights, Camera, Action."

Projection Transform

The final step is to take the 3D scene and project the scene onto a 2D plane so it can be drawn on the screen. This is called projection space or clip space.

To move from view space to projection space, a projection matrix is used to transform the geometry. For more information on the different types of project matrices, see Chapter 5.

Backface Culling

To cull something means to remove something that is useless. When drawing real-time computer graphics, speed is always a consideration. The 3D graphics' pipeline is designed to be fast and efficient. One of the optimizations that occurs in the graphics pipeline is to remove triangles that face away from the viewer in a process called backface culling.

In most cases, when drawing triangles, they can be seen only from one side. When looking at a teapot made of triangles, you would not want to draw the triangles that were on the far side facing away from you. These triangles are covered by the closer front-facing triangles on the front of the teapot.

So the question is how does the graphics pipeline know which triangles are front facing and which are not? This is set by something called winding order. By default, triangle vertices are defined in clockwise order in XNA Game Studio. Counter-clockwise triangles are culled and do not continue to other operations down the graphics pipeline.

You might not want triangles' backfaces culled for things such as a road sign. The `GraphicsDevice.RasterizerState` enables you to set `CullMode` values to turn off backface culling or change it to cull clockwise triangles.

Viewport Clipping

The front-facing vertices of the triangles still contain four values for X, Y, Z, and the homogeneous W component. Before continuing further down the graphics pipeline, the X, Y, and Z components of the vertex are divided by the homogeneous W component to return the vertex to three dimensions. This process is called the homogeneous divide.

The next optimization is to determine whether any part of the triangle is visable on the screen. Triangle vertices with values of X and Y between –1 to 1 and a Z value of 0 to 1 are visible on the screen and should continue down the graphics pipeline.

Triangles that are partly on the screen are clipped. This means that only the visible part of the triangle continues to the next stage of the pipeline while the part that is off the screen is culled away. Triangles that don't have any viable part on the screen are culled entirely.

> **Note**
>
> Although the graphics hardware removes geometry that is not visible, it is often a performance for games with several objects to do some type of higher level culling, such as frustum culling, before drawing objects to the scene.

Rasterization

The next stage, rasterisation, is the process of converting the triangles into the individual pixels that are used on the screen. Triangles don't always fall directly over a pixel. If a triangle covers the center of the pixel, then the pixel is considered covered by the triangle.

> **Note**
>
> There are several complex rules over which pixel is shaded by which triangle and how to determine which triangle to use when two triangle edges share the same pixel. This is a more advanced topic that is not covered in this book. In most cases, it won't matter to you.

Pixel Shader

The triangles have now been broken down into a set of pixels. Each of these pixels are now sent to the pixel shader. The pixel shader is a small program run on the graphics hardware that takes the pixel's position and determines the pixels output color. The output color can be anything from a solid color to a complex color calculated from lighting values.

The pixel shader is programmable and controlled by creating a small pixel program that is compiled and set to the graphics hardware. Creating your own custom pixel shaders are covered in Chapter 8.

Pixel Tests

Although you calculated the output pixel color from the pixel shader, it can still be thrown away. There are several pixel tests that could cause the pixel to be invalid and not used.

Scissor

The scissor test enables you to set a rectangle region within the screen to draw to. If you want to limit rendering to a small section of the screen, you can set the `GraphicsDevice.ScissorRectangle` property to a rectangle within the screen bounds and set the `ScissorTestEnable` property of the `GraphicsDevice.RasterizerState` to true. By default, the scissor test is not enabled.

Stencil

The next test is called the stencil test. Stencil tests utilize a special buffer called the stencil buffer, which, if enabled, contains eight bits of data per pixel on the screen. When drawing geometry, the stencil value can be incremented, decremented, or not changed. This value can then be used to test future pixels to determine whether they pass the stencil test.

In most common cases, you do not change the stencil values or use the stencil test when drawing objects on the screen. Stencil tests are used for more advanced rendering techniques like stencil shadows and portal rendering.

Depth

The last test and one of the most important is the depth test. The depth test utilizes a special buffer called the depth buffer also called a Z buffer, which can contain 16 or 24 bits of data per pixel. The depth test is used to determine whether a pixel is in front or in back of other pixel's draw to the same location.

Consider the case where you draw the teapot on the table. If you look down on the teapot and draw an imaginary line through the teapot and through the table, it hits a number of triangles for a number of pixels.

How do you determine which triangle is in front of which? A depth buffer works by storing the depth Z value of the last triangle drawn to a specific pixel. When the next triangle that uses the same pixel is about to be drawn, the depth value of that triangle is compared to the value stored in the depth buffer. If the value of the new triangle is lower, the pixel color continues down the graphics pipeline and the new Z value is stored in the depth buffer. If the value is higher, the new pixel is behind a pixel that was already drawn and can be thrown out if the depth test is enabled.

Blending

Although the final color for the pixels that make up the triangles being drawn are determined, this does not mean this is the desired color to be draw on the screen. A number of blending operations can occur between the source color from the pixel shader and any colors already draw on the screen.

For more information on the different blending operations, see Chapter 7, "States, Blending, and Textures."

Final Output

The final step in the pipeline is to store the color value. In most cases, this is stored on the screen in what is called the backbuffer, which is discussed later in this chapter.

It is also possible to draw directly onto a texture using a render target. After you complete a drawing to a render target, use it like any other texture in your game. Render targets are discussed in Chapter 7.

Reach and HiDef Graphics Profiles

Before we talk about how to utilize the XNA Game Studio graphic types to draw our first triangles on the screen, we need to talk about one of the new changes that XNA Game Studio brings to graphics.

Graphics Profiles Define Platform Capabilities

In the beginning of this chapter, we discuss how developing 3D games can be complex. One of the large complexities has to do with how to handle different graphics hardware on different platforms. Different graphics cards support slightly different capabilities. This is especially a problem on the PC where there are many different types of graphics cards. In the past, developers had to query the graphics card for its capabilities and then make decisions about what types of graphics operations to use based on those stated capabilities. There were hundreds of capabilities and they were difficult to test because you needed all of the different types of graphics hardware accessible to you.

To solve this problem, XNA Game Studio 4.0 introduced the concept of a `GraphicsProfile`. A graphics profile sets specific requirements that a particular piece of hardware meets or it doesn't. If a particular piece of hardware supports a profile, then all of the features in that profile are supported. As a developer, you set the target `GraphicsProfile`. XNA Game Studio 4.0 supports two graphics profiles called `Reach` and `HiDef`.

When your application starts on a machine that does not support every aspect of that profile, a NoSuitableGraphicsDeviceException exception is thrown and enables you to inform the user that he or she can't run your game.

At development time, the graphics APIs in XNA Game Studio enforce the `GraphicsProfile` you selected. For example, the `Reach` profile supports textures with width and height up to 2048 while the `HiDef` profile supports up to 4096. If your game targets the `Reach` profile, an exception is raised if you attempt to create a texture larger than 2048 even if the PC you are developing on could support the `HiDef` profile. Having the APIs respect the `GraphicsProfile` setting ensures you do not use features that are not supported and your game runs on any platform that supports the `Reach` profile.

The Reach Profile

As the name implies, the `Reach` profile provides the capability to give your game title reach meaning that the code and content you create for your game would work on a large number of platforms and systems. The `Reach` profile is supported by Windows Phone 7 devices, the Xbox 360, and Windows PCs with a DirectX 9 GPU that support shader model 2.0 or higher.

The HiDef Profile

The `HiDef` profile was designed for systems that provide more advanced graphics capabilities. This limits the number of platforms and systems that support the profile but allow for a richer set of capabilities. The `HiDef` profile is a superset of the `Reach` profile meaning that any platform and system that can support the `HiDef` profile by definition also supports the `Reach` profile. The `HiDef` profile is supported on the Xbox 360 and Windows PCs with a DirectX 10 or equivalent GPU.

So what does equivalent mean? It means that some DirectX 9 hardware is also supported but must meet some specific capabilities. The list of capabilities is too long and of little importance to cover in this book.

> **Note**
>
> A chart comparing the differences between the profiles including specific details such as texture size limitations can be found in the appendix.

Determining GraphicsProfile Support

To find out whether your graphics card supports either of these two profiles, create a simple Windows application and call the `GraphicsAdapter.IsProfileSupported` method passing in a specific graphics profile.

Let the 3D Rendering Start

Now that we covered the basics of the graphics pipeline and some of the math that is involved, let's see how these concepts relate to the types exposed by XNA Game Studio and draw some triangles on the screen.

GraphicsAdapter

Use the `GraphicsAdapter` class to enumerate and update graphics adapters. Most single monitor PCs have one graphics adapter that represents their physical graphics card and the one connection it has to the monitor. Some graphics cards provide two graphics adapters so you can plug two monitors into a single graphics card. In this case, the `GraphicsAdapter` allows for selecting two different adapters so you can display the graphics on two different monitors. The Xbox 360 and Windows Phone devices provide only one `GraphicsAdapter`.

The static `GraphicsAdapter.DefaultAdapter` property returns the default adapter in which there are multiple adapters to chose from. For example, when you set the left monitor to be the primary display in Windows, the `DefaultAdapter` property returns the `GraphicsAdapter` to display graphics on the left monitor.

The static `GraphicsAdapter.Adapters` property returns a read-only collection of the adapters available. You rarely need to enumerate graphics adapters. Most commercial PC games provide a settings page that enables the user to select the adapter so he or she can change which display to view the game. This is required when the game runs in full screen and the user does not have the normal window controls to click and drag.

If your game uses windowed mode, the user can click and draw the window to the other monitor. Although this changes the `GraphicsAdapter` used, all of this is handled for the user by XNA Game Studio by default.

> **Note**
>
> If you developed games using a native graphics API before and tried to support monitor dragging and toggling between windowed and full-screen modes, you know the trouble to get it working properly.

The `GraphicsAdapter` provides several properties and methods that enable you to determine what type of graphics card is available and what the capabilities of the graphics card are.

The `QueryBackBufferFormat` and `QueryRenderTargetFormat` methods can be used to determine whether a specific back buffer or render target format is supported. We discuss back buffers and render targets later in this chapter.

The `DeviceName`, `Description`, `DeviceId`, `VendorId`, `SubSystemId`, and `Revision` properties are used to return information about a specific `GraphicsAdapter` to determine whether the graphics card is a specific model from a specific manufacturer. They also enable you to determine what the current driver version is.

> **Note**
>
> A graphics driver is a piece of software that enables the operating system to communicate with the graphics hardware. Graphics card manufacturers create the driver to instruct the operating system on how to perform specific tasks with the hardware.

Each `GraphicsAdapter` can support a number of display modes. Just as you can set your PC to use different resolutions, your game can use different resolutions. The `SupportedDisplayModes` returns a collection of `DisplayMode` structures. Each structure contains properties for the `Width`, `Height`, and `Format` for the display mode. A display format is defied using the `SurfaceFormat` enumeration. The format defines how each pixel on the screen is represented in terms of the data that is stored at each pixel. The most common format, `SurfaceFormat.Color,` is a 32-bit value that stores four channels of red, green, blue, and the alpha each with 8 bits of value.

> **Note**
>
> Alpha is a term used to express how transparent a color is. We cover transparency and alpha in Chapter 7.

The `DisplayMode` structure also provides two helper properties called `AspectRatio` and `TitleSafeArea`. The `AspectRatio` is the value you get when you divide the width of the display by the height of the display. This is often calculated when working with 3D graphics and is sometimes calculated incorrectly. The property gives you a simple way to retrieve the value without having to calculate the value each time you need it.

When drawing to televisions, they have outer edges of the screen cut off. If you draw to the top left corner of the screen and display this on a number of televisions, you will notice that each television has a slightly different position of where the top left corner is. The `TitleSafeArea` defines where you should limit drawing game specific graphics such as text and the heads up display for a character. Using the property ensures that your graphics are visible across different televisions.

GraphicsDevice

The `GraphicsDevice` in your game is responsible for issuing the drawing commands down to the driver, which then goes through the graphics pipeline. The `GraphicsDevice` is also responsible for loading graphics resources such as textures and shaders.

Every `GraphicsDevice` contains a back buffer. The back buffer is the data buffer where the graphics that ultimately end up displayed on your monitor are drawn to. It is called the back buffer because it is not displayed on the screen. What you see displayed on the screen is actually the contents of what is called the front buffer. Drawing occurs to the back buffer, so you don't see each piece of geometry appear on the screen as it is drawn. After all of the draw calls are performed in a single frame, the `GraphicsDevice.Present` method is called, which flips the front and back buffers to display the contents of the back buffer.

Along with a width and height, the back buffer also has a `SurfaceFormat`, which defines the bit layout and use for each pixel in the back buffer. As we discussed, the `GraphicsAdapter` can be used to determine what formats are available. The most common is the `Color` format.

If a back buffer resolution that is smaller than the display is requested on the Xbox 360 or on a Windows Phone, the final image is up scaled to fit the screen. If the aspect ratio is different, then the display black bars are displayed on the top and bottom of the screen called letterboxing or on the sides of the screen called pillarboxing to fill out the extra space.

When calling the `Present` method, you specify a `PresentInterval`. The present interval is used to determine when the flip of the front and back buffers occur. Displays on computer monitors, televisions, and phone screens all have a refresh rate. The refresh rate is the speed that the displays update their physical screen commonly between 30Hrz to 60Hrz. When displays update, they do so by updating line by line until the entire screen is updated. At that time, the display performs a vertical retrace sometimes referred to as a vblank. This is when the screen is moving from the end of where it is currently updating to the start again. If the front and back buffers are flipped in the middle of the display, updating a graphical artifact called tearing can occur. Tearing is where part of the displayed screen shows one frame while another portion displays another. This occurs if the flip is not in sync with the vertical retrace because you change what is drawn on the screen in the middle of the display updating the display. To prevent this, the flip should occur during the vertical retrace.

When you set the `PresentInterval`, you have three options of when the flip between the front and back buffers occurs. If `PresentInterval.One` is used, the buffers wait to flip until the display is in the vertical retrace phase. This means the fastest your graphics can update is the same speed as your display's refresh rate. `PresentInterval.Two` is used only to flip the buffers every other vertical retrace. `PresentInterval.Immediate` can be used to perform the flip as soon as possible and not wait for the vertical retrace. Although your game can suffer from tearing artifacts, this mode is helpful when performing performance analysis because it does not limit the speed at which your game can draw.

Along with the back buffer, a `GraphicsDevice` can also contain another buffer called the depth buffer. The depth buffer is used to store the depth values of a triangle when the color of that triangle is written to the back buffer. The values are used in the depth test position of the graphics pipeline that we discussed previously. A depth buffer can have three types of formats specified by the `DepthFormat` enumeration. The first is `Depth16`, which stores a 16-bit floating point value at each pixel. `Depth24` provides a 24-bit floating point number to store depth at each pixel. Finally, the `Depth24Stencil8` format provides the same 24 bits for depth but also enables another 8 bits to be used by the stencil buffer. The stencil buffer is used in the stencil test portion of the graphics pipeline.

Creating the GraphicsDevice

Although it is possible to create the graphics device yourself using the `GraphicsDevice` constructor, this is not needed when using `Game` class because the default template creates a `GraphicsDeviceManager`, which creates the `GraphicsDevice`.

If you want to change any of the default values that the `GraphicsDeviceManager` uses to create the graphics device, you can use code similar to the following in your `Game` class constructor.

```
public Game1()
{
    // Create new graphics device manager that will create the graphics device
    graphics = new GraphicsDeviceManager(this);
    // Set the width and height we would like for the back buffer
    graphics.PreferredBackBufferHeight = 800;
    graphics.PreferredBackBufferWidth = 480;
    // Set the back buffer format to color
    graphics.PreferredBackBufferFormat = SurfaceFormat.Color;
    // Set the depth format to have depth and stencil values
    graphics.PreferredDepthStencilFormat = DepthFormat.Depth24Stencil8;
    // We don’t want to use multisampling
    graphics.PreferMultiSampling = false;

    Content.RootDirectory = "Content";
}
```

The `GraphicsDeviceManager.SupportedOrientations` can be used to define which orientations you want your game to support. This is useful for Windows Phone games where the user can flip the physical phone device and might do so because of the requirements of the game. If you elect to support an orientation when the user rotates the phone, the screen flips to support the new orientation. For example, if you want to support a landscape game where the user can play the game by rotating the phone to the left or right, use the following lines of code:

```
graphics.SupportedOrientations = DisplayOrientation.LandscapeLeft |
                                 DisplayOrientation.LandscapeRight;
```

Now when the user rotates the phone from one horizontal position to the other, the display automatically flips and your game appears correctly for the new orientation.

Reference Devices

There is a special type of graphics device called a reference device also known as ref device. A reference device does not use the graphics hardware and implements all of the graphics functionally in software on the computer's CPU. A reference device can be used when the graphics hardware does not support some specific graphics feature. The downside is that the reference device is tremendously slow compared to graphics hardware.

Drawing with Primitives

If you have never developed 3D graphics applications, this is where you become a graphics developer for the first time. All of the geometry that we draw in future chapters builds upon the following primitive drawing.

Primitive Types

XNA Game Studio supports four types of primitives defined by the `PrimitiveType` enumeration: `TriangleList`, `TriangleStrip`, `LineList`, and `LineStrip`. A `TriangleList` is a list of triangles just like its name implies. Every three vertices are treated as a new triangle meaning that each triangle requires three vertices to be stored. In most cases, this is overkill because most triangles are directly next to another triangle. With a `TriangleStrip`, the last three vertices are used to draw the next triangle. For example, you can draw two triangles with only four vertices. The first three vertices make up the first triangle and the last vertex is used in conjunction with the last two vertices from the first triangle to form the second triangle. Using triangle strips helps reduce the amount of memory required to store the vertices of triangles. The `TriangleStrip` is by far the most common `PrimitiveType` used.

The ListList is similar to the triangle list except that only two points are needed to form a single line. Every two points creates a new line segment. The LineStrip is similar to a triangle strip except that only the last point from the previous line segment is needed to form another line.

Vertex Types

Each vertex that makes up the triangle contains the position data for where the triangle is in space, but it can also contain other data such as the color that the triangle should be, texture coordinates for textured triangles, or normal data for lighting calculations. XNA Game Studio provides several ways to build in vertex types that can be used when defining your triangles or lines. You can also create your own vertex types for more complex rendering scenarios.

Drawing Primitives

There are ways to render geometric primitives in XNA Game Studio. All four methods are provided by the `GraphicsDevice` class. The first two methods—`DrawUserPrimitives` and `DrawUserIndexedPrimitives`—both work by specifying an array of vertices and a primitive type to the `GraphicsDevice`, which, in turn, draws the primitives.

The second two methods—`DrawPrimitives` and `DrawIndexedPrimitives`—use vertices that have been stored on the graphics hardware. Storing vertex data directly on the graphics hardware lowers drawing latency because the data needed to draw the geometry is already in memory in the graphics hardware near where the computations are occurring.

Along with vertex data, the two indexed methods—`DrawUserIndexedPrimitives` and `DrawIndexedPrimitives`—also use index values that are used to index into the vertex data. Instead of using the vertex data directly to draw the primitive, indexed vertices are defined using an offset into the vertex data. This saves space when multiple primitives use the same vertex.

DrawUserPrimitives

The easiest way to draw a primitive on the screen is to declare an array of vertices and pass these to the `DrawUserPrimitive` method. Along with the vertices, we create an instance of `BasicEffect`. We cover effects and how they work in Chapter 6, "Built-In Shader Effects." For now, you know that the effect is needed so the graphics hardware knows what to do in the vertex shader and pixel shader portions of the graphics pipeline.

Define the following member variables in your `Game` class:

```
VertexPositionColor[] userPrimitives;
BasicEffect basicEffect;
```

Next, define the vertices and create a new instance of `BasicEffect` and set some of the properties. Add the following lines of code in your `Game` class `LoadContent` method:

```
// Create the verticies for our triangle
userPrimitives = new VertexPositionColor[3];
userPrimitives[0] = new VertexPositionColor();
userPrimitives[0].Position = new Vector3(0, 1, 0);
userPrimitives[0].Color = Color.Red;
userPrimitives[1] = new VertexPositionColor();
userPrimitives[1].Position = new Vector3(1, -1, 0);
userPrimitives[1].Color = Color.Green;
userPrimitives[2] = new VertexPositionColor();
userPrimitives[2].Position = new Vector3(-1, -1, 0);
userPrimitives[2].Color = Color.Blue;

// Create new basic effect and properites
basicEffect = new BasicEffect(GraphicsDevice);
basicEffect.World = Matrix.Identity;
basicEffect.View = Matrix.CreateLookAt(new Vector3(0, 0, 3),
                                       new Vector3(0, 0, 0),
                                       new Vector3(0, 1, 0));
basicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                              GraphicsDevice.Viewport.AspectRatio,
                                              0.1f, 100.0f);
basicEffect.VertexColorEnabled = true;
```

In the previous code, you define three vertices of type `VertexPositionColor`. This vertex type holds a position and color for each vertex. You define the position and color for the top, right, and left vertices. The order you specify these is important because order is not culled by default in XNA Game Studio.

After you create your vertices, you create an instance of `BasicEffect`. The World, View, and Projection properties are used in the vertex shader portion of the graphics pipeline to transform the input vertices to draw on the screen. For now, don't worry about the specific values for these matrices. The `VertexColorEnabled` property is set to true to tell the `BasicEffect` to use the color from each vertex as the output value of the pixel shader.

The final step is to issue the draw call using the `DrawUserPrimitives` method. Add the following lines of code to your `Game` class `Draw` method:

```
// Start using the BasicEffect
basicEffect.CurrentTechnique.Passes[0].Apply();
// Draw the primitives
GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleList,
                                                       userPrimitives, 0, 1);
```

Before you can call `DrawUserPrimitives`, tell the `GraphicsDevice` to use the `BasicEffect`. This is done by calling Apply on the `EffectPass` you use. EffectPass is covered in Chapter 8, "Introduction to Custom Effects."

Next, call `DrawUserPrimitives`. This method is generic and expects the vertex type as the generic `T` type parameter. In this case, pass `VertexPositionColor` as the vertex type. The first parameter defines what `PrimitiveType` you want to draw. Specify `TriangleList` to draw a list of triangles. The second parameter is an array of vertices of the same type used for the generic `T` parameter. Pass the `userPrimitives` array you created previously. The third parameter is used to specify an offset into the array of vertices. If your array contains multiple primitive vertices and you want to draw only a subset of those, you can specify an offset into the source array. For your purpose, you don't want any offset, so set the value to 0. The final parameter `DrawUserPrimitives` takes the total number of primitives to draw. If your array defines multiple primitives, specify that number to draw with this parameter. For this example, you draw only a single triangle, so specify a value of 1.

If you build and run the previous example, you should see a single triangle like that in Figure 4.22. Notice that each vertex has its own color. and the color of the middle sections of the triangle change across the triangle to each vertex. This is called interpolation and occurs on values that are specified per vertex. Because the color is specified per vertex, the different vertex colors have to interpolate over the pixels between the two vertices.

Now let's update the example to draw multiple primitives using the `LineStrip` primitive. First, update where the vertices are created with the following lines of code:

```
// Create the verticies for our lines
userPrimitives = new VertexPositionColor[4];
userPrimitives[0] = new VertexPositionColor();
```

```
userPrimitives[0].Position = new Vector3(-1, 1, 0);
userPrimitives[0].Color = Color.White;
userPrimitives[1] = new VertexPositionColor();
userPrimitives[1].Position = new Vector3(-1, -1, 0);
userPrimitives[1].Color = Color.White;
userPrimitives[2] = new VertexPositionColor();
userPrimitives[2].Position = new Vector3(1, 1, 0);
userPrimitives[2].Color = Color.White;
userPrimitives[3] = new VertexPositionColor();
userPrimitives[3].Position = new Vector3(1, -1, 0);
userPrimitives[3].Color = Color.White;
```

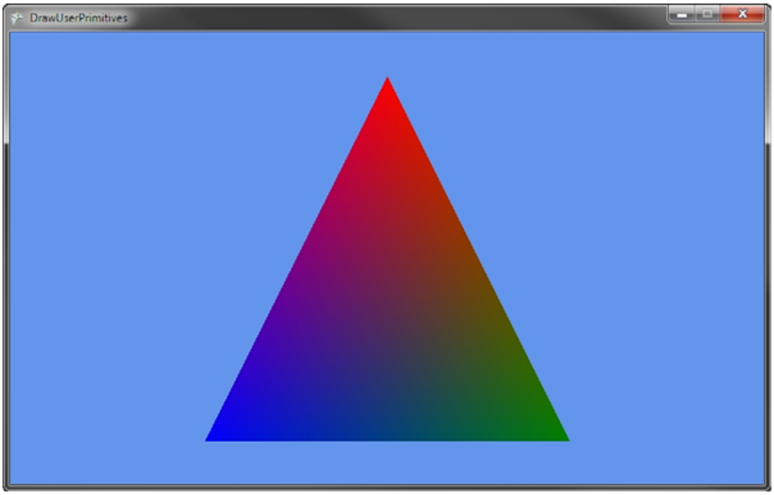


Figure 4.22 Using **`DrawUserPrimitives`** to draw a single triangle

> **Note**
>
> The lines are set to `Color.White` to help make them more visible. You can use any color just like you would for triangles.

Then, update your `DrawUserPrimitives` call to specify the `LineStrip` primitive type and the new number of primitives:

```
// Draw the primitives
GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.LineStrip,
                                                        userPrimitives, 0, 3);
```

If you run the example now, it should look like Figure 4.23.

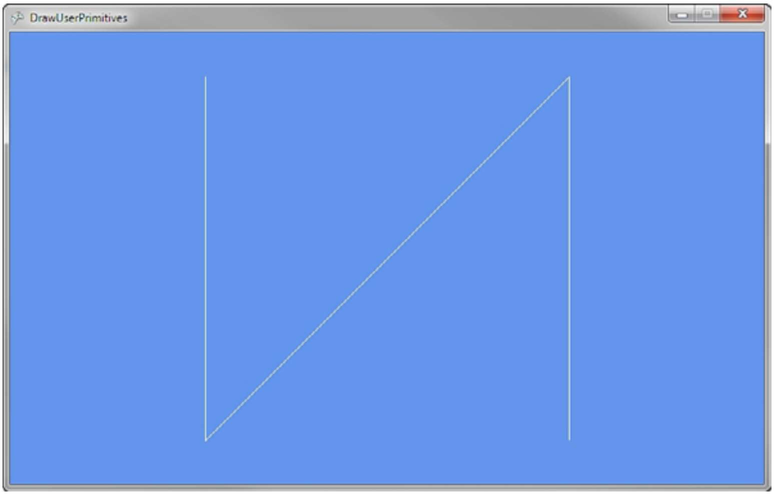


Figure 4.23 Using **`DrawUserPrimitives`** to draw a **`LineStrip`**

DrawUserIndexedPrimitives

If your primitives commonly use the same vertices for a number of different primitives, it can save you memory to use indexed primitives instead. To draw a user indexed primitive, you need to specify an array of indexes to use when drawing the triangles. These index values are used to look up each specific vertex to use when rendering the primitive.

To draw a user indexed primitive, update the current example. Add an array of index value. These are integer values that represent offsets into the vertex array you also pass into the `DrawUserIndexedPrimitives` method. Update the member variables for your Game class to have the additional index array like the following code:

```
VertexPositionColor[] userPrimitives;
short[] userPrimitivesIndices;
BasicEffect basicEffect;
```

Use a 16-bit `short` array to store the index values. Because each index is represented by a `short`, the index values can be between only 0 and 65535, which is the maximum positive number a `short` can represent. You can also use an array of 32-bit `int` values, which can store positive values of more than four billion but this has two repercussions. The first is that storing index values as `int` values means that each index takes up two more bytes than when using shorts thus doubling the memory usage for your index values. The other is that 32-bit `int` indexes are supported only in the `HiDef GraphicsProfile`, so your game must usea `HiDef GraphicsDevice`.

Next, create the vertices and index values for your primitives. In this case, draw a square, often called a quad, that is made of two triangles. Update the existing example's `LoadContent` method to contain the following code:

```
// Create the verticies for our triangle
userPrimitives = new VertexPositionColor[4];
userPrimitives[0] = new VertexPositionColor();
userPrimitives[0].Position = new Vector3(-1, 1, 0);
userPrimitives[0].Color = Color.Red;
userPrimitives[1] = new VertexPositionColor();
userPrimitives[1].Position = new Vector3(1, 1, 0);
userPrimitives[1].Color = Color.Green;
userPrimitives[2] = new VertexPositionColor();
userPrimitives[2].Position = new Vector3(-1, -1, 0);
userPrimitives[2].Color = Color.Blue;
userPrimitives[3] = new VertexPositionColor();
userPrimitives[3].Position = new Vector3(1, -1, 0);
userPrimitives[3].Color = Color.Purple;

// Create the indices used for each triangle
userPrimitivesIndices = new short[6];
// First Triangle
userPrimitivesIndices[0] = 0;
userPrimitivesIndices[1] = 1;
userPrimitivesIndices[2] = 2;
// Second Trianglel
userPrimitivesIndices[3] = 1;
userPrimitivesIndices[4] = 3;
userPrimitivesIndices[5] = 2;
```

Like the previous example, create an array of `VertexPositionColor` to represent the vertices in your primitives. The code creates four vertices: one for each corner of the quad. A `short` array is then created to specify which vertex to use for each primitive. The first triangle uses the top left, top right, and bottom left vertices making sure to define them in clockwise order. The second triangle uses the top right, bottom right, and bottom left.

Finally, update your `Game` class `Draw` method to use the `DrawUserIndexPrimitives` method:

```
// Draw the primitives
GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionColor>
➥(PrimitiveType.TriangleList,
                                                    userPrimitives, 0, 4,
                                                    userPrimitivesIndices, 0, 2);
```

Like the previous example with `DrawUserPrimitives`, the `DrawUserIndexedPrimitives` call is generic and you pass the `VertexPositionColor` as the generic `T` type parameter. The first parameter is again the primitive type. The second parameter is again the array of vertices. The third parameter is the vertex offset that should be used. For index primitives, this offset is added to each index value to obtain the final index to use in the vertex array. The fourth parameter is the number of vertices that is used in the draw call. In this case, use four vertices. This parameter is often the size of the vertex array itself unless you use offsets. The fifth parameter is the array used to store the index values. For the example, that is the `userPrimitivesIndices` variable. The sixth parameter is the offset into the index array that should be used. In this case, you don't need to offset, so the value is 0. The final parameter is again the number of primitives to draw. For the example, you try two triangles, so a value of 2 is specified.

If you build and run the example code, you should see an image like that in Figure 4.24 where a four-color quad is draw on the screen.

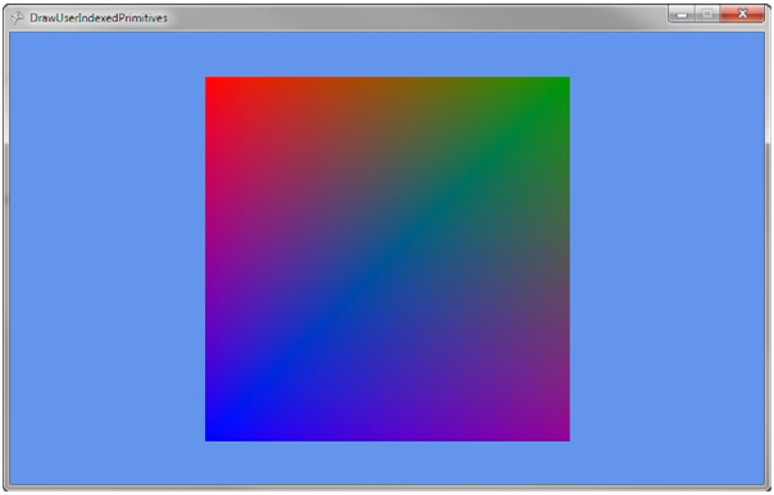


Figure 4.24 Using **`DrawUserIndexedPrimitives`** to draw a quad

Let's lower the memory used by the index array by updating the example to just use four index values and use the `TriangleStrip PrimitiveType`. Update where you create the index array to look like the following:

```
// Create the indices used for each triangle
userPrimitivesIndices = new short[6];
// First triangle
userPrimitivesIndices[0] = 0;
userPrimitivesIndices[1] = 1;
userPrimitivesIndices[2] = 2;
// Second Triangle
userPrimitivesIndices[3] = 3;
```

The second triangle is formed by using the last two index values plus the new additional value. In this case, the triangle is formed by `userPrimitivesIndices[3]`, `userPrimitivesIndices[2]`, and `userPrimitivesIndices[1]`. This second triangle saves 32 bits of data by not using two additional index values.

The `DrawUserIndexedPrimitives` method call needs to be updated to use only `PrimitiveType.TriangleStrip` and not `TriangleList`. If you run the example code now, you will not see any visible difference yet you are saving valuable memory space. It is not always possible to save space and use triangle strips; in these cases, you should use `TriangleLists`.

DrawPrimitives

Unlike the user versions of the draw primitives methods that use arrays to store the vertex data, `DrawPrimitive` uses a `VertexBuffer` to store the vertices for your primitives.

A `VertexBuffer` stores the vertices in the graphics cards hardware memory where they can be quickly accessed. Unlike when using arrays, a `VertexBuffer` does not require the vertices to be sent from the main system memory to the graphics card each time you want to draw. Sending large amounts of data from the main system memory to the graphics card is not a fast operation and can slow down your rendering speed. So for larger geometry, use a `VertexBuffer` to store the vertex data and to call the `DrawPrimitives` method.

First, create two member variables in your `Game` class: one for the `VertexBuffer` and another for the `BasicEffect`. Add the following lines of code to create the member variables:

```
VertexBuffer vertexBuffer;
BasicEffect basicEffect;
```

Next, create the `VertexBuffer` and set the data that is stored inside it. Add the following lines of code to your `Game` class `LoadContent` method:

```
basicEffect = new BasicEffect(GraphicsDevice);
basicEffect.World = Matrix.Identity;
basicEffect.View = Matrix.CreateLookAt(new Vector3(0, 0, 3),
                                       new Vector3(0, 0, 0),
                                       new Vector3(0, 1, 0));
basicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                              GraphicsDevice.Viewport.AspectRatio,
                                              0.1f, 100.0f);
basicEffect.VertexColorEnabled = true;

VertexPositionColor[] vertexData = new VertexPositionColor[3];
vertexData[0] = new VertexPositionColor();
vertexData[0].Position = new Vector3(0, 1, 0);
vertexData[0].Color = Color.Red;
vertexData[1] = new VertexPositionColor();
vertexData[1].Position = new Vector3(1, -1, 0);
```

```
vertexData[1].Color = Color.Green;
vertexData[2] = new VertexPositionColor();
vertexData[2].Position = new Vector3(-1, -1, 0);
vertexData[2].Color = Color.Blue;

// Create our VertexBuffer
vertexBuffer = new VertexBuffer(GraphicsDevice,
                                typeof(VertexPositionColor), 3,
                                BufferUsage.None);
vertexBuffer.SetData<VertexPositionColor>(vertexData);
```

As with the previous example, you first create an instance of `BasicEffect` and set some of the properties for the transforms and enable vertex coloring.

Again, declare an array of `VertexPositionColor` and define three vertices. Use this array to set the data to store in the `VertexBuffer`.

Next, create the instance of the `VertexBuffer`. The `VertexBuffer` takes the `GraphicsDevice` to store the data as the first parameter. Use the `GraphicsDevice` property of the `Game` class to use the default `GraphicsDevice` that is created when your game starts. The second parameter takes the `Type` of the vertex that is to be stored in the `VertexBuffer`. Because you want to store vertices of type `VertexPositionColor`, specify the type using the `typeof` operator. The third parameter is the number of vertices the `VertexBuffer` is going to hold. In this case, there are only three vertices. The fourth and final parameter is the `BufferUsage`. In cases where you know you will never need to read the contents of the `VertexBuffer`, you can specify `BufferUsage.WriteOnly`, which causes the `VertexBuffer` to throw an exception if you ever try to read back any data. Using `WriteOnly` enables the graphics hardware to chose specific memory locations in the hardware that allow for more efficient drawing.

After the `VertexBuffer` is created, the `SetData` method is called to send the vertex data to the graphics hardware. `SetData` is a generic method that takes the type of vertex data as the generic `T` parameter. The array of vertices is then passed as the lone parameter. The size of the `VertexBuffer` must be large enough to store the amount of vertices stored in the array that is passed in. Overloaded versions of `SetData` are available that enable you to specify an offset, start index, number of vertices to set, and the size of each vertex. These overloads are useful when the source vertex array is slit into multiple vertex buffers.

The final steps you need to draw are to tell the `GraphicsDevice` the `VertexBuffer` you plan to use next and to call the `DrawPrimitives` method. Add the following lines of code to your `Game` class `Draw` method:

```
// Set which vertex buffer to use
GraphicsDevice.SetVertexBuffer(vertexBuffer);
// Set which effect to use
basicEffect.CurrentTechnique.Passes[0].Apply();
// Draw the triangle using the vertex buffer
GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
```

The `GraphicsDevice` is told which `VertexBuffer` to use by calling the `SetVertexBuffer` method and passing in the one you created. This is the `VertexBuffer` that is used when any `DrawPrimitives` or `DrawIndexPrimitives` methods are called until another `VertexBuffer` is set on the `GraphicsDevice`. Overloads of SetVertexBuffer are available that enable an offset into the `VertexBuffer` to be specified.

Next, the `EffectPass.Apply` method is called like in the previous example.

Finally, call the `DrawPrimitives` method. The first parameter is the `PrimitiveType`. The second parameter is the start vertex. This is the vertex in the buffer to start drawing from. This value is combined with any offset value set when `SetVertexBuffer` is called. The final parameter is the number of primitives to draw, which you set to 1 because you need to draw only the single triangle.

If you run the previous example code, you see output similar to Figure 4.25. Notice how this example and the first example that used `DrawUserPrimitives` produced the same results but use different functions to draw the primitives.

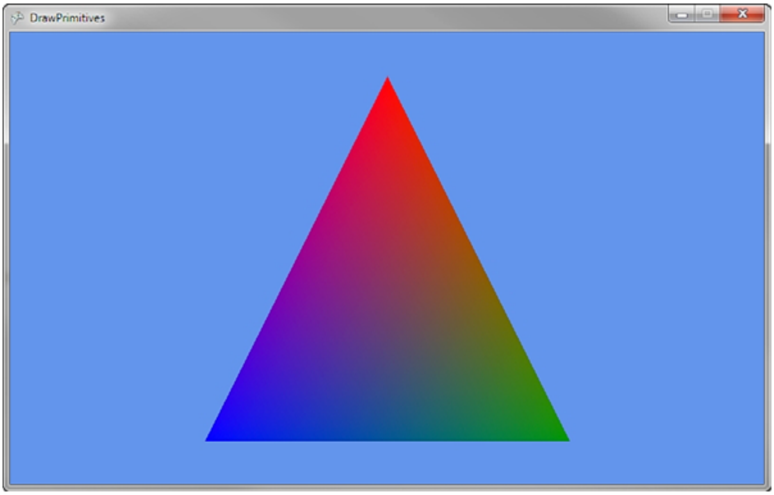


Figure 4.25 Using **`DrawPrimitives`** to draw a triangle

DrawIndexedPrimitives

The final draw primitives called `DrawIndexedPrimitives` uses both a `VertexBuffer` and an `IndexBuffer`. Like a `VertexBuffer`, the `IndexBuffer` stores data using the graphics hardware's memory so it can be quickly accessed by the GPU when drawing. The `IndexBuffer` stores index values like the ones you used previously when using the `DrawUserIndexPrimitives` method.

To draw indexed primitives using an index buffer, add a member variable to your Game class to hold the instances of the `IndexBuffer`, `VertexBuffer`, and `BasicEffect`.

```
VertexBuffer vertexBuffer;
IndexBuffer indexBuffer;
BasicEffect basicEffect;
```

Next, create the index data to store in the `IndexBuffer` and create the instance of the `IndexBuffer`. Also create the `VertexBuffer` and `BasicEffect` as you did previously:

```
basicEffect = new BasicEffect(GraphicsDevice);
basicEffect.World = Matrix.Identity;
basicEffect.View = Matrix.CreateLookAt(new Vector3(0, 0, 3),
                                       new Vector3(0, 0, 0),
                                       new Vector3(0, 1, 0));
basicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                            GraphicsDevice.Viewport.AspectRatio,
                                            0.1f, 100.0f);
basicEffect.VertexColorEnabled = true;

VertexPositionColor[] vertexData = new VertexPositionColor[4];
vertexData[0] = new VertexPositionColor();
vertexData[0].Position = new Vector3(-1, 1, 0);
vertexData[0].Color = Color.Red;
vertexData[1] = new VertexPositionColor();
vertexData[1].Position = new Vector3(1, 1, 0);
vertexData[1].Color = Color.Green;
vertexData[2] = new VertexPositionColor();
vertexData[2].Position = new Vector3(-1, -1, 0);
vertexData[2].Color = Color.Blue;
vertexData[3] = new VertexPositionColor();
vertexData[3].Position = new Vector3(1, -1, 0);
vertexData[3].Color = Color.Purple;

// Create our VertexBuffer
vertexBuffer = new VertexBuffer(GraphicsDevice, typeof(VertexPositionColor),
                                4, BufferUsage.None);
vertexBuffer.SetData<VertexPositionColor>(vertexData);

short[] indexData = new short[4];
// First triangle
indexData[0] = 0;
indexData[1] = 1;
indexData[2] = 2;
// Second triangle
indexData[3] = 3;

// Create our IndexBuffer
indexBuffer = new IndexBuffer(GraphicsDevice, IndexElementSize.SixteenBits,
                              4, BufferUsage.None);
indexBuffer.SetData<short>(indexData);
```

In the previous code, you create an array of `shorts` to store the index values. Because you use a triangle strip, you need only four index values. The `IndexBuffer` is then created. The first parameter is the `GraphicsDevice` where the `IndexBuffer` is stored. The second parameter is the size each element in the index buffer is. Remember that index values can be 16-bit `short` values or 32-bit `int` values. You select `IndexElementSize.SixteenBits` to specify that 16-bit index values. The fourth parameter is the number of index values the `IndexBuffer` holds. You specify 4 the size of your index data array. The last parameter is `BufferUsage`, which is similar to the same parameter on the `VertexBuffer`. You specify `None`, which will enables read back if necessary later.

Finally, tell the `GraphicsDevice` which `IndexBuffer` to use when the `DrawIndexPrimitives` method is called. Add the following lines of code to your `Game` class `Draw` method:

```
// Set which vertex buffer to use
GraphicsDevice.SetVertexBuffer(vertexBuffer);
// Set which index buffer to use
GraphicsDevice.Indices = indexBuffer;
// Set which effect to use
basicEffect.CurrentTechnique.Passes[0].Apply();
// Draw the triangle using the vertex buffer and index buffer
GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleStrip, 0, 0, 4, 0, 2);
```

As with the previous examples, you set the `VertexBuffer` to use and call Apply on the `EffectPass` you want to use. You also specify the Indices parameter of the `GraphicsDevice` setting its value to your `IndexBuffer`. The `DrawIndexedPrimitives` method is then called to draw a quad on the screen. Like all of the other draw methods, the first parameter is the `PrimitveType`, which, in this case, is a `TriangleStrip`. The second parameter is the base vertex, which is an offset that is added to each of the values in the index buffer. The third parameter is the minimum vertex index of all of the vertices used. The fourth parameter is the number of vertices that are going to be used when drawing the triangles. The fifth parameter is the starting index to use in the index buffer. The last parameter like the other draw methods is the number of primitives to draw.

If you run the example code, you should see a colored quad similar to the one in Figure 4.26.

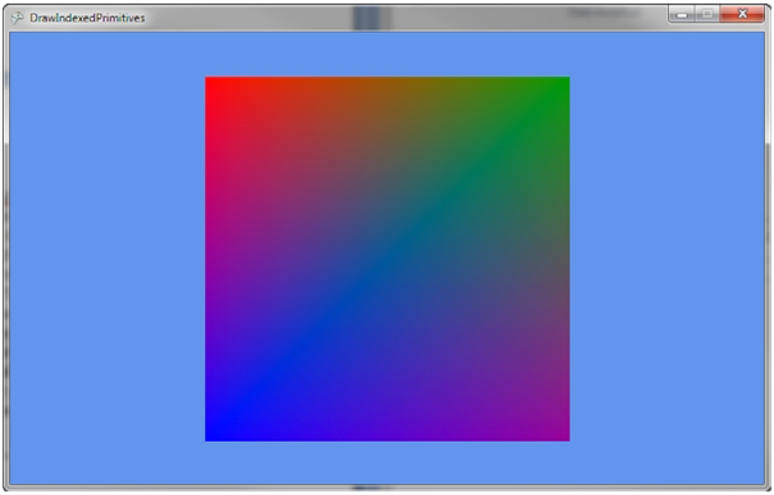


Figure 4.26 Using `DrawIndexedPrimitives` to draw a quad

Summary

It is time to take a deep breath. We covered a lot of information about 3D graphics in this chapter including what 3D graphics is in regards to interactive games and applications, some of the basic mathematics needed to work with 3D graphics, an overview of the graphics pipeline, and finally how to draw some geometry on the screen using the four main drawing methods provided by XNA Game Studio.

This page intentionally left blank

5

Lights, Camera, Action!

As you move forward with your learning about graphics in this book, you need to understand how your "scene" (the things rendered in your game) is visualized. Much like a director working on a movie, you are responsible for dictating how your world is visualized. You use a camera to do this, just as a director would. You also explore easier ways to store the objects that make up your worlds. In this chapter, you learn about:

- Controlling the camera in your 3D scenes
- Rendering models

Well, actually, lights and action are described later, but one out of three isn't a bad start. When people speak about cameras in three-dimensional (3D) games, they rarely speak about a rendered camera, but more the spot in which your viewpoint comes from. Even the name of the matrix describing this is the view matrix. Now is the time to take a look at the thing called the camera.

Why Do I See What I See?

Much like you'd expect, a camera in a 3D graphics scene is basically the same thing as a camera in real life. It has a position where the camera is in the world, and it takes a picture of the world to show you on a flat surface. In 3D graphics, it is the same thing; the camera dictates how you see the world.

Of the three most common matrices you set when rendering, two of them are used to control the camera. The world matrix is used to define the world and is unrelated to the camera, but the view matrix and the projection matrix are each used to modify how the camera is either positioned or the properties it uses to render. Let's look at a quick example.

First, you need to create a new Windows Game project and add the box.fbx model to your content project. Models are discussed later in this chapter, but for now, load it and render it to show camera interaction. You need to add a new variable to render this model, so add the following:

```
Model model;
```

Then, update your `LoadContent` method to instantiate this object:

```
model = Content.Load<Model>("box");
(model.Meshes[0].Effects[0] as BasicEffect).EnableDefaultLighting();
```

Ignore the second line for now, as it is discussed more in Chapter 6, "Built-In Shader Effects." It is being added right now so the scene doesn't look flat and ugly. Naturally, the first line loads the model (which you probably guessed was a box). Finally, you need to render the box using a camera, so add the following to your `Draw` method after the call to `Clear`:

```
model.Draw(Matrix.Identity,
Matrix.CreateLookAt(new Vector3(2, 3, -5), Vector3.Zero, Vector3.Up),
Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f));
```

Running this application shows you a box rendering in the center of your window, such as on the one you see in Figure 5.1.

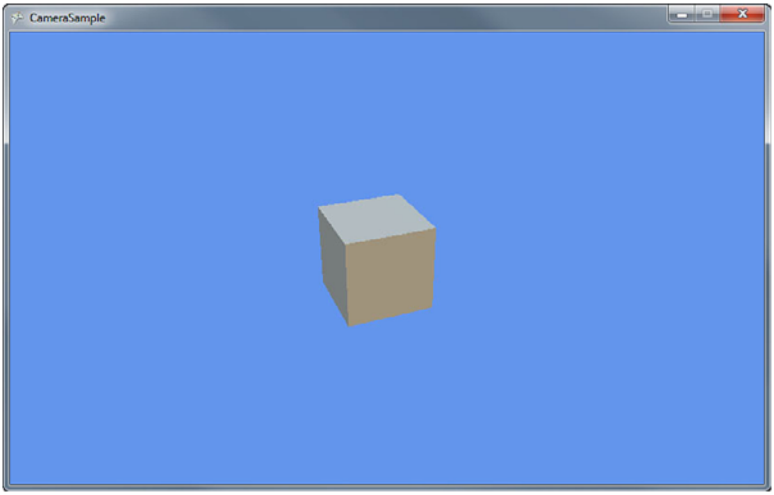


Figure 5.1 A simple scene's camera rendering

The camera in that scene is defined by the `CreateLookAt` method and the `CreatePerspectiveFieldOfView` method. They define how the final image of your scene looks and the position, properties, and lens type of the camera. The `CreateLookAt` method returns the view matrix, whereas the other returns the projection matrix. Let's look at those in more depth.

View Matrix

The view matrix is the easiest to visually conceptualize, as it has only three components: the position of the camera in world space, where the camera looks, and what direction is up. In the example you used a moment ago, the camera is located at a point (2,3,–5) in world space, looking at the origin (0,0,0), with the up direction being the constant up direction on the Vector class (0,1,0); you can visualize this as seen in Figure 5.2.

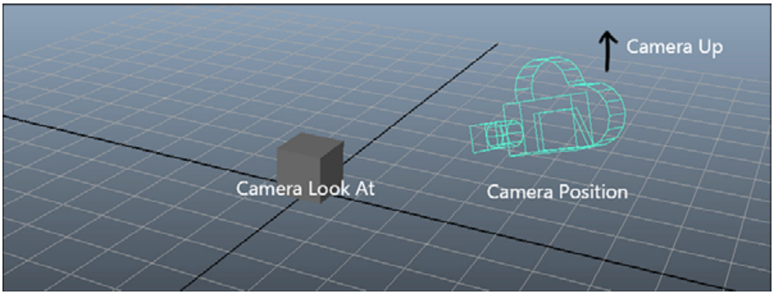


Figure 5.2 How the camera views the scene

Notice how you see a portion of the cube? What if you moved the camera somewhere else to see a different portion of the cube? Try replacing that first vector with a new `Vector3(3,-2,4)` and see how your view changes. Notce that you're looking at a completely different angle of the cube, as seen in Figure 5.3.

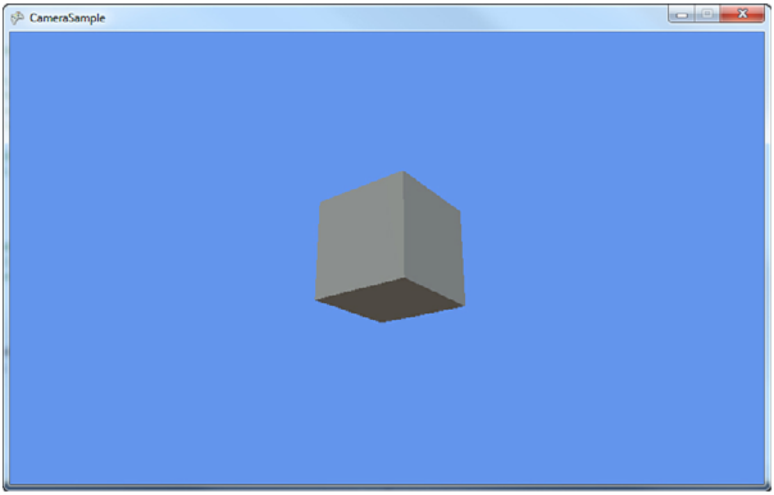


Figure 5.3 How the camera views the scene from another angle

The second parameter to `CreateLookAt` is also easy to conceptualize because it is the position the camera looks toward. In the example given, the box is located at (0,0,0), and the camera looks there, too, so the box is in the center of the screen. If you changed the second parameter instead to a vector of (–2,0,0), you would expect that the camera would be looking more to the left (negative x is to the left when the camera is in the positive Z axis), so the box should be more to the right. By extension, using a vector of (2,0,0) would make the box appear to the left (because the camera looks to the right of it).

Of course, if you changed the location of the camera to the opposite side of the Z axis (3,–2,4), then the look at point of (2,0,0) would cause the box to appear on the right because the camera looks to the left of it (positive x is to the left when the camera is in the negative Z axis).

The final parameter to the `CreateLookAt` method is the vector that describes what up is. In the previous example, you used `Vector3.Up`, which is a vector of (0,1,0), which means the up direction of the camera follows the positive Y axis. Imagine the camera is your eyes (because after all, that is what it is), and you're standing on a flat surface. The positive Y axis (0,1,0), is up to you. If you stood on your head, up to you would be the negative Y axis, and everything would appear upside down. Change your last parameter in the `CreateLookAt` method to `Vector3.Down`, and notice that you're still looking at the box in the same way; it's just upside down now, much like it would be if you looked at it while standing on your head.

By the same mechanism, if you tilted your head to the right, so up was `Vector3.Right`, you'd still see the same image, but instead of being right-side up or upside down, it would be rotated 90 degrees. You can visualize the up vector as the imaginary ray that runs from the center of the camera through the top of the camera, as you can see in Figure 5.4.

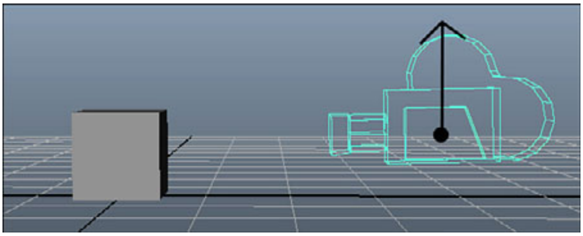


Figure 5.4 Which way is up?

Projection Matrix

Compared to the view matrix, the projection matrix is much more difficult to explain! When thinking of the view matrix as the camera, you can think of the projection matrix as the lens. There are many more lens types than cameras!

The method we used in the example earlier to describe our projection matrix was `CreatePerspectiveFieldOfView`, which creates a perspective projection matrix, based on the field of view. As was discussed in Chapter 4, "Introduction to 3D Graphics," perspective is what makes things closer appear larger, whereas things further away appear smaller. There are two major types of projection matrices you can create—perspective and orthographic—both of which are discussed next.

Perspective

The perspective matrix lets you define the viewing frustum. To visualize what this is, imagine a pyramid. The tip of the pyramid is the location of the camera (or your eye). Anything that exists inside that pyramid can be seen; anything that exists outside of the pyramid cannot be seen. The various methods to create perspective projection matrices are used to define the size and shape of this pyramid. See Figure 5.5.

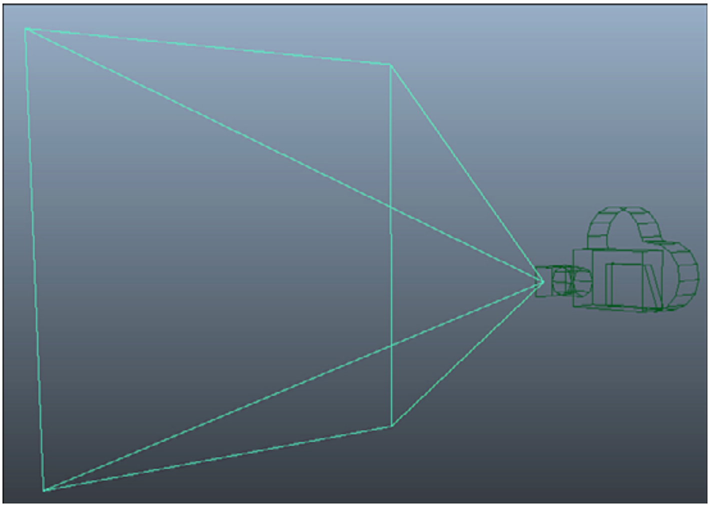


Figure 5.5 The viewing frustum

First, let's look at the matrix used in the example. It was created using the `CreatePerspectiveFieldOfView` method, which has a few parameters. The first is the field of view parameter, the second is the aspect ratio, and the last two are the near and far planes, but what do these actually mean? The field of view is the angle that the top of the pyramid forms and is specified in radians. In the example, we used an angle of 45 degrees, which is somewhat common. If you used 90 degrees (`MathHelper.PiOver2`) instead, you'd have a much larger field of view, and thus, a larger viewing area. This would cause your rendered objects to appear smaller because the same screen area would have to render a larger world area (see Figure 5.6 for more information on the math of the field of view or fov).

The second parameter is the aspect ratio, which is simply the width of your scene divided by the height of your scene. Notice here that you use the `Viewport` property of the `GraphicsDevice` and you use the `AspectRatio` property from that. This is the most common aspect ratio you use, but if you render to something other than the back buffer (say a render target), you might want to render at a different aspect ratio. The aspect ratio here is just like the aspect ratio you see on televisions. Standard-definition televisions have a 1.33 (4/3) aspect ratio, whereas most high-definition televisions have a 1.77 (16/9) aspect ratio. Most of the old resolutions you've seen in computers were 1.33 (4/3) aspect ratios (640×480, 800×600, 1024×768, and so on).

> ### Aspect Ratio, Integers, and Floats
>
> Aspect ratio is defined as a float value. If you are calculating your aspect ratio using the width divided by the height, and your width and height parameters are both integers, so be sure to cast at least one of them to float before the division. If you do not, the compiler does the division as integers and casts the final value to float, which isn't what you want. For example, if you try to calculate the aspect ratio of a standard definition television set, you use the width of 4 divided by the height of 3, and in integer math, 4/3 = 1. It would then cast the 1 to the float of 1.0f, which is incorrect. If you cast one of the values to float before the division, though, you get 4.0f/3 = 1.3333f, which is the value you want.

The last two parameters are the near and the far planes of the frustum. Anything closer than the near plane or anything farther than the far plane is not visible. You can think of the far plane as the base of the pyramid, whereas the near plane is where you would "cut off" the top of the pyramid.

See Figure 5.6 for more information on the math for a field of view projection matrix.

Using the field of view is one way to create a perspective projection matrix, but not the only way. There are two other helper methods you can use, namely `CreatePerspective` and `CreatePerspectiveOffCenter`. Each of these creates your pyramid in slightly different ways, but before we get to that, let's modify the example to draw a lot of boxes so it's easier to see how the changes behave. Replace your draw code with the following (this might seem way more complicated than it is) to see a series of boxes like you see in Figure 5.7:

```
protected override void Draw(GameTime gameTime)
{
    const int numberBoxes = 3;
    const int radiusMultiple = numberBoxes + 1;
    GraphicsDevice.Clear(Color.CornflowerBlue);
    float radius = model.Meshes[0].BoundingSphere.Radius;
Matrix view = Matrix.CreateLookAt(
        new Vector3(0, radius * radiusMultiple,
            radius * (radiusMultiple * radiusMultiple)),
        new Vector3((numberBoxes / 2) * (radius * radiusMultiple) - 1,
              (numberBoxes / 2) * (radius * radiusMultiple) - 1, 0),
             Vector3.Up);
Matrix proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
         GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
    for (int x = 0; x < numberBoxes; x++)
    {
        for (int y = 0; y < numberBoxes; y++)
        {
            Vector3 pos = new Vector3((y * (radius * radiusMultiple)) - 1,
                (x * (radius * radiusMultiple)) - 1, -(y + x));
            model.Draw(Matrix.CreateTranslation(pos), view, proj);
        }
    }
    base.Draw(gameTime);
}
```

Although this seems more complicated than it is, all you're doing here is drawing nine different boxes. They're rendered in a square pattern with each one rendered at a different spot in the world and at a variety of different depths. You should notice how ones that are farther away are slightly smaller. The radius calculation is discussed in more detail later this chapter when we talk about the models. The rest is just simple math to position the boxes around the screen.

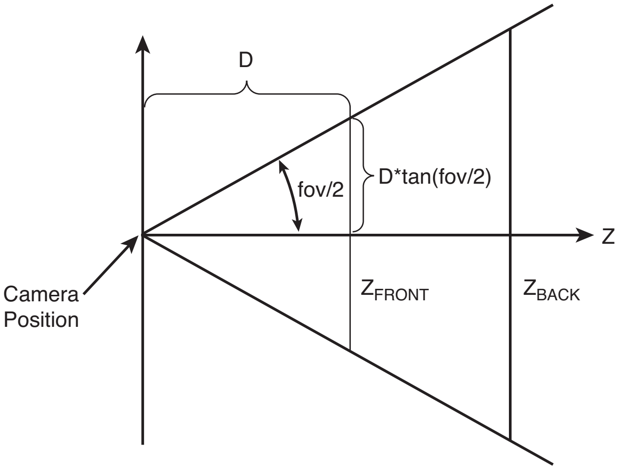


Figure 5.6 The math behind field of view

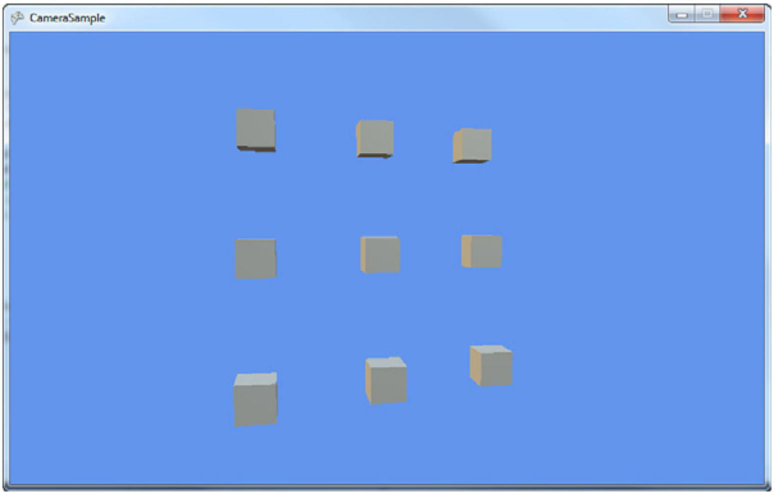


Figure 5.7 The view of a bunch of boxes

The `CreatePerspective` method takes only four parameters to describe what your pyramid should look like. The first one is the width of the top of the pyramid (where the near plane cuts it off), and the second is the height of the same. The third parameter is the distance to the near plane, and the fourth is the distance to the far plane (or the base of the pyramid). Modify your matrix creation as follows:

```
Matrix proj = Matrix.CreatePerspective(1.0f, 0.5f, 1.0f, 100.0f);
```

If you run the program now, notice that the picture has changed somewhat dramatically. All nine boxes don't even appear fully onscreen anymore! This is because you changed the shape of your viewing frustum (or the pyramid), and portions of those boxes are now outside of it. See Figure 5.8.

The `CreatePerspective` method assumes that you are looking at the center point of the near plane (formed by the width and height), but that isn't a requirement either. You can use the `CreatePerspectiveOffCenter` method, which is similar to the nonoff center method. Rather than a width and height being passed in, you instead must pass in the left, right, top, and bottom positions. For example, if you use the following instead of the `CreatePerspective` method you used earlier, you would get the same output:

```
proj = Matrix.CreatePerspectiveOffCenter(-0.5f, 0.5f, -0.25f, 0.25f, 1.0f,
➥100.0f);
```

This is because you have the same width and height, and they are centered. Using this method enables you even more control over the location and size of the viewing frustum.

There is also another common type of projection matrix, the orthographic projection.

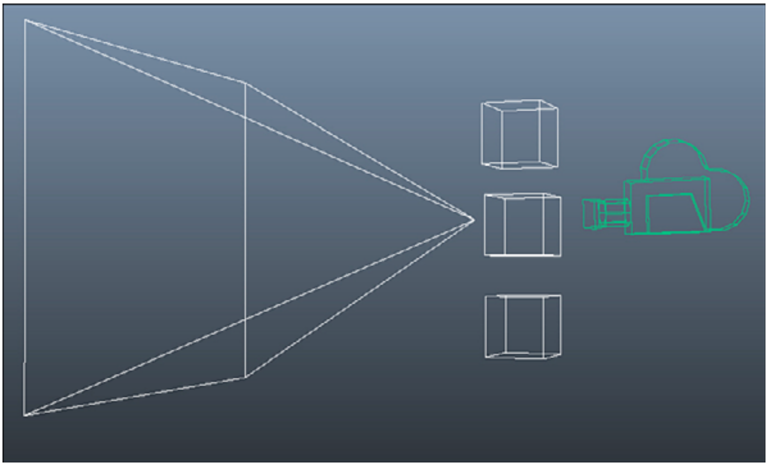


Figure 5.8 Modifying the view frustum directly

Orthographic

Much like a perspective matrix, an orthographic matrix builds a viewing frustum, but instead of a pyramid shaped structure, it is more rectangular. The viewing frustum does not get larger between the near plane and the far plane, and no perspective foreshortening occurs. All objects of the same size are rendered the same size, regardless of how far they are from the camera.

There are two helper methods to create these types of projection matrix: `CreateOrthographic` and `CreateOrthographicOffCenter`. Much like the perspective counterparts, these describe the orthographic volume, with the former being centered and the latter capable of being off center. If you replaced your project matrix with the following, you would see all nine boxes, but they'd all appear on the same plane:

```
proj = Matrix.CreateOrthographic(15.0f, 15.5f, 1.0f, 100.0f);
```

With the basics of projection matrices and view matrices out of the way, now you can actually create some camera types!

Camera Types

Although any individual game probably has only a single type of camera, there are a wide variety that can be used—cameras that follow the players, cameras that never move, and cameras that change depending on the circumstances...the possibilities are endless.

Static Cameras

The easiest camera to think about is a static camera. As the name implies, a static camera doesn't move. It sits there and just looks out. The camera's you've been using in this chapter so far are simple static cameras. Let's create a helper class that is your camera type. Right-click your project, select Add->New Item, choose Game Component (calling it `CameraComponent.cs`) and add the following variables to it:

```
protected Matrix view;
protected Matrix proj;
```

You need a way to get these values back from your component, so add two property getters for them now:

```
public Matrix View
{
    get
    {
        return view;
    }
}
public Matrix Projection
{
    get
    {
        return proj;
    }
}
```

You also need to initialize these to something reasonable, so in your `Initialize` overload, add the following:

```
view = Matrix.CreateLookAt(new Vector3(0, 0,-16),
Vector3.Zero, Vector3.Up);
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    Game.GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
```

With that, you've made yourself a fixed static camera. Let's modify the game to use this camera instead. Add a new variable to your `Game` class to hold the camera:

```
CameraComponent camera;
```

Then, in your game's Initialize overload, add this component to your game's component list:

```
Components.Add(camera = new CameraComponent(this));
```

Then finally, switch your call to `Draw` on the model to use this camera component:

```
model.Draw(Matrix.CreateTranslation(pos), camera.View, camera.Projection);
```

If you run the application now, you see a slightly different view because your camera is in a different location. This camera can be extended to support different types of cameras, which you should experiment with.

Models

Earlier in this chapter, you were likened to a director, using your camera to bring your world to life. Just like the directors in movies who often use models to be the stars in their films, you will do the same thing!

What Is a Model?

A *model* is essentially a collection of geometry that is rendered together to form an object in the world. You've used them throughout this chapter already, but let's take more time now to go through the `Model` class itself, which has quite a bit of data. You remember from last chapter when you were rendering things with vertex buffers and index buffers? Models use these objects, too, and are a great way to hold geometry.

If you use a Digital Content Creation (DCC) package, such as SoftImage Mod Tool (see Figure 5.9), to create your objects, when you export those creations to a file (such as an .fbx file or an .x file), you can add these to your content projects to use the Model content importer. These are loaded into the `Model` class.

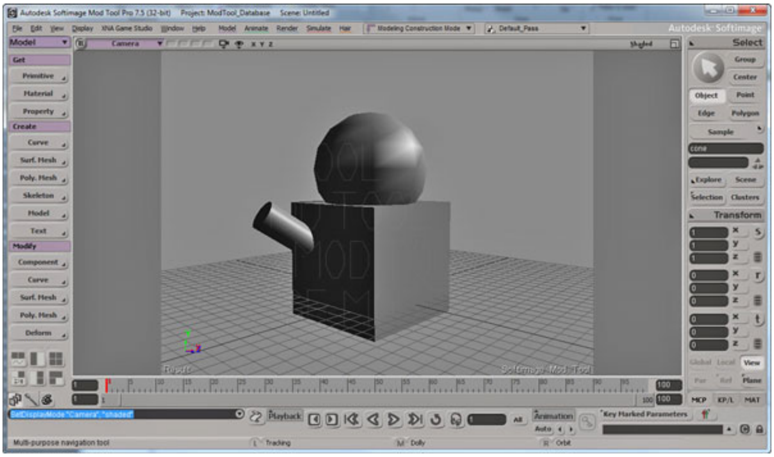

Autodesk Softimage Mod Tool Pro 7.5 (32-bit) Project: ModTool_Database Scene: Untitled
File Edit View Display XNA Game Studio Window Help Model Animate Render Simulate Hair
Modeling Construction Mode
Default_Pass
Autodesk Softimage
Model
Camera
Shaded
Get
Primitive
Material
Property
Create
Curve
Surf. Mesh
Poly. Mesh
Skeleton
Model
Text
Modify
Component
Curve
Surf. Mesh
Poly. Mesh
Deform
Select
Group
Center
Object
Point
Edge
Polygon
Sample
cone
Explore
Scene
Selection
Clusters
Transform
View
Plane
MCP
KP/L
MAT
Playback
Animation
Auto
Key Marked Parameters
SetDisplayMode "Camera", "shaded"
Multi-purpose navigation tool
Tracking
Dolly
Orbit

Figure 5.9 A DCC package creating a model

Models

The first method you can look at on the Model class is the `Draw` method, because you've already used it in this chapter. This method (as the name implies) draws the model using the supplied transform with the provided view and projection matrices and with the defaults for everything else. It is a quick and easy way to show something onscreen as you've seen so far.

A `Tag` property can also be used to store any type of data. Many content importers also include extra information in this object for the game to use. Let's take a look at the `Meshes` property next.

Meshes

The `Meshes` property returns a collection of `ModelMesh` objects. In the box you've been using up until now in this chapter, each model has only had one mesh inside of it. Although that is the simplest form of a model, models can be extremely complex. Imagine a model that represents a city. The city would have roads, buildings, and signs. Each of these objects can be represented by a different model mesh, whereas the combined set of meshes is the full model.

Each mesh also has a `Draw` method that can be used to render the mesh separately, but notice that this method has no parameters. This does the rendering of the mesh, but it does so with the current settings of the effects rather than setting the world, view, and projection matrices that the model's `Draw` method does. This is because there is extra information inherent to each mesh that might change where it is in the world; we discuss this later in this chapter.

Another piece of information found on the mesh is the `BoundingSphere` property, which describes a sphere that encompasses the entire mesh. The bounding sphere object has a few fields and methods that are used for a variety of reasons. The `Center` and `Radius` fields are used to describe the sphere, whereas there are a few helper methods such as `Contains` and `Intersects` used to detect objects in the sphere. You can also `Transform` the sphere using that helper method.

Each mesh can also have a `Name` (that can be set by your code or set during content importing and loading) and `Tag` similarly to the model itself. Aside from the bones (which are discussed in a moment), the last two properties on the mesh are `Effects` and `MeshParts,` which are interrelated. The `Effects` collection is the collection of effects that is associated with each mesh part and controls how the mesh parts are rendered. The number of effects and mesh parts is identical, and you can update or modify the effects for any of the mesh parts you'd like.

> **Note**
>
> Effects are discussed in depth in Chapers 6 and 8, "Built-In Shader Effects," and "Introduction to Custom Effects."

Each mesh part is the portion of the model that is actually rendered. Notice that one of the properties it has is the `Effect` that this part uses. Changing this property also affects the `Effects` property on the mesh at the same index this mesh part exists at. The rest of the properties contain the information about the geometry and gives you the information you need to draw the mesh part with the `DrawIndexedPrimitives` method. You can see the `IndexBuffer` and `VertexBuffer` properties that hold the index and vertex data and the `PrimitiveCount` to dictate how many triangles to draw. When drawing data using this information, you can always use a primitive type `PrimitiveType.TriangleList`.

A few other pieces of information on the mesh part can be used when setting the vertex buffer and index buffer to the device, such as `NumVertices` (which is the number of vertices in the buffer), `VertexOffset` (which is the number of vertices to skip before getting to the vertices required for this mesh part), and `StartIndex` (which is the number of indices to skip before getting to the first index required by this mesh part). Like the `Model` and `ModelMesh` before it, this also includes a `Tag` property.

Normally each mesh part is separated from the other portions of the mesh based on the material it uses to render itself. You see the term material used often in DCC packages and throughout literature on this subject. For the purposes of this book, you can consider the material to be the combination of the `Effect` used to render the geometry and the textures required to do so.

For example, in the imaginary city model, you can also imagine that the road portions of the model are all one mesh. If you had three different kinds of roads, such as a paved street, a gravel road, and a dirt road, you would potentially use different textures and possibly even different effects for those, so your roads mesh would have three different mesh parts: one for the paved street, one for the gravel roads, and a final one for the dirt roads.

See Figure 5.10 for an example of how models, meshes, and mesh parts are all interrelated.

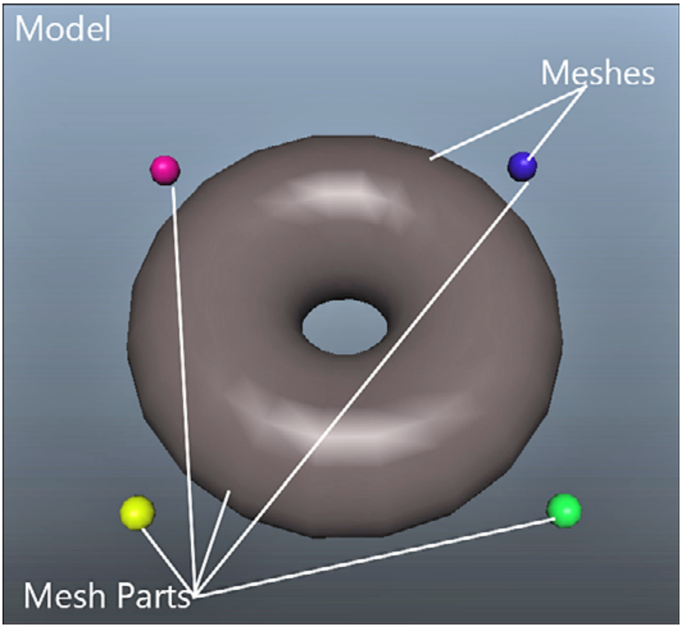


Figure 5.10 Models, meshes, and mesh parts

Throughout the discussions of models, you didn't about one particular type of object, so let's take a look at it now.

Bones

The `ModelMesh` has a property `ParentBone` that you skipped, and the model itself had quite a few methods and properties that deal with bones. What exactly are bones (aside from things like a femur or a song by Alice in Chains)? At a high level, bones are what connect each mesh to every other mesh. See Figure 5.11 for an example of a simple model with three meshes.

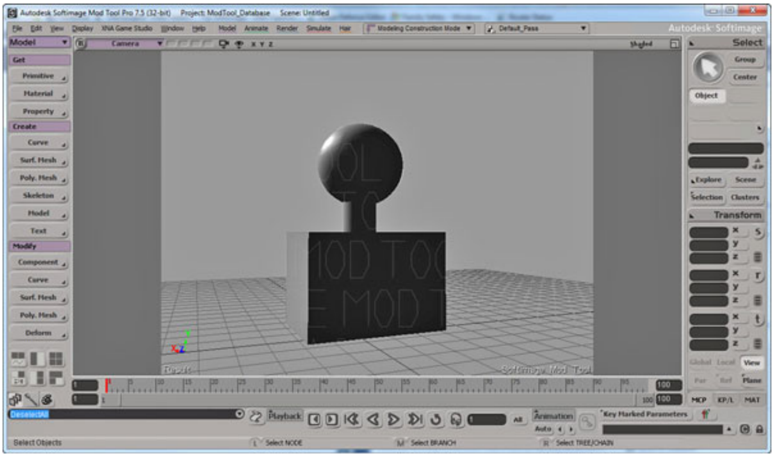

Autodesk Softimage Mod Tool Pro 7.5 (32-bit) Project: ModTool_Database Scene: Untitled
Model
Camera
Get
Primitive
Material
Property
Create
Curve
Surf. Mesh
Poly. Mesh
Skeleton
Model
Text
Modify
Component
Curve
Surf. Mesh
Poly. Mesh
Deform
Select
Group
Center
Object
Explore
Scene
Selection
Clusters
Transform
Global
Local
View
Plane
MCP
KP/L
MAT
Playback
Animation
Auto
Key Marked Parameters
Select Objects
Select NODE
Select BRANCH
Select TREE/CHAIN

Figure 5.11 A model with three meshes

Let's assume you want to build a model of a person, and what is in Figure 5.11 is what you have so far. You have the sphere representing the head, a cylinder representing the neck, and a cube representing the torso. Each of these body parts can be considered a mesh in the larger full model, and bones are what tie these together. Each bone tells you the relationship between itself, its parent, and its children.

One of the meshes in the model is the root bone. This is the mesh that is the root of the other meshes and is accessed via the `Root` property on the model, which returns a `ModelBone` object. These objects have a few different properties to help explain the relationship of that particular bone to the others in the model. First, it has the `Children` property that is a collection of `ModelBone` objects that define the children of this bone. You can also get the `Parent` (again, as a `ModelBone`) of this bone. Like the mesh, you can also get the `Name` of this bone, which can be set during content import and creation.

You also see the `Index` property, which is the index of this bone in the model's `Bones` collection (which is the entire collection of bones for the model). Finally, you see the `Transform` property, which is the transform of this bone in relation to its parent. Let's look at an example.

In the model in Figure 5.11, you can imagine that the torso would be the root bone with a single child being the cylinder forming the neck, which in turn, has a single child as the sphere forming the head. If the model had arms and the torso was the root, you could extrapolate out that it would have multiple children for the neck, arms, and legs.

Now, assume that the torso has a transformation of `Matrix.Identity`. It has no scale or rotation and it is located at the origin (0,0,0). However, the neck is approximately three units above that, so the `Transform` property of the neck bone is `Matrix.CreateTranslation(0,3,0)`. You can see that the head is even higher, yet approximately three units above its parent, the neck, so its `Transform` property would be `Matrix.CreateTranslation(0,3,0)`, even though it is approximately six units above the torso! If your head was slightly turned, you could instead have its `Transform` property be `Matrix.CreateRotationX( MathHelper.PiOver4) * Matrix.CreateTranslation (0,3,0)`. The transform is always based on the parent bone and not the model itself. This is especially useful for animated characters because like in this example, you can easily have your head turn side to side by simply changing the rotation transform on a single bone, rather than repositioning the entire set of geometry.

There are also three methods on the `Model` class you can use to get or set these transforms. You can use the `CopyBoneTransformsTo` method to create a copy of the transforms each bone has into a new array of matrices (for example, what you might want to pass into an effect). You can also use the similar (but opposite order) method of `CopyBoneTransformsFrom` to update all the bones with these new transforms. There is also another helper method called `CopyAbsoluteBoneTransformsTo`, which like the nonabsolute one, copies all the transforms into an array of matrices. The difference is this one combines each child's transform with all of its parent transforms.

What this means, in the previous example, by using `CopyBoneTransformsTo` you would have an array of three matrices: the first member having `Matrix.Identity` and the second two members having `Matrix.CreateTranslation(0,3,0)`. However, if you use `CopyAbsoluteBoneTransformsTo`, you would have the same array of three matrices with the first two members remaining the same, but the third member would instead be `Matrix.CreateTranslation(0,6,0)` because it combines the transform with its parents transforms.

Rendering Models

Enough of all this text—'let's actually use the model class to draw some stuff onscreen! Create a new Windows Game project and add any model you'd like from the downloadable examples. Now, get ready to do some rendering. You need to add a variable for your model, such as:

```
Model model;
```

Next, update your `LoadContent` method to get the object instantiated:

```
model = Content.Load<Model>("YourModel");
foreach (ModelMesh mm in model.Meshes)
{
    foreach (Effect e in mm.Effects)
    {

IEffectLights iel = e as IEffectLights;
        if (iel != null)
        {
            iel.EnableDefaultLighting();
        }
    }
}
```

Naturally, you need to replace the YourModel with whatever model you picked. Although you haven't gotten to the effects yet, the next section of code enables default lights for all portions of the model that have it. This is discussed in more detail in Chapter 6, but for this discussion, it makes your models look nice.

> **Note**
>
> By default, models loaded through the content pipeline have default effects set, depending on the materials set in the DCC package used to create it. A common effect used is `BasicEffect`, which is discussed in Chapter 6.

Next, because you've picked any random model you wanted, and models can be of a wide variety of sizes, you need to add a helper method to get the size of the largest mesh in the model, so you can scale it correctly. In a real game, you never have to do this trickery because you control the scale and size of the models, but here, it's a quick operation to have a good guess. Add this private function to your game class:

```
private float GetMaxMeshRadius(Model m)
{
    float radius = 0.0f;
    foreach (ModelMesh mm in m.Meshes)
    {
        if (mm.BoundingSphere.Radius > radius)
        {
            radius = mm.BoundingSphere.Radius;
        }
    }
    return radius;
}
```

This goes through each mesh and finds the one with the largest bounding radius and returns the largest radius it finds. Now you can add another helper method to draw the model, which you' update a few times over the next few pages to see the various ways of rendering the model. The first one is the easiest:

```
private void DrawModel(Model m, float radius, Matrix proj, Matrix view)
{
    m.Draw(Matrix.CreateScale(1.0f / radius), view, proj);
}
```

The only thing interesting you use for the model is the world matrix. You create a scale that makes the model approximately 1.0f units. This enables you to use a static camera regardless of model size and still see it in its entirety.

Finally, you need to do the actual rendering. Replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    float radius = GetMaxMeshRadius(model);
Matrix proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
        GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
Matrix view = Matrix.CreateLookAt(new Vector3(2, 3, 4), Vector3.Zero, Vector3.Up);
    DrawModel(model, radius, proj, view);
    base.Draw(gameTime);
}
```

Notice here that you're positioning the camera slightly offset from the model and four units away. This is possible only because of the scaling done in the `DrawModel` call; otherwise, you have to base your camera size on the size of the model. Nothing here is exactly new, though, and running the application shows your model rendered on screen.

Next, let's add the new method `DrawModelViaMeshes` to use the meshes instead of the single `Draw` method on model. Add the following method and update your `DrawModel` call to use this one instead:

```
private void DrawModelViaMeshes(Model m, float radius, Matrix proj, Matrix view)
{

Matrix world = Matrix.CreateScale(1.0f / radius);
    foreach (ModelMesh mm in model.Meshes)
    {
        foreach (Effect e in mm.Effects)
        {
            IEffectMatrices iem = e as IEffectMatrices;
            if (iem != null)
            {
                iem.World = GetParentTransform(m, mm.ParentBone) * world;
                iem.Projection = proj;
                iem.View = view;
            }
        }
        mm.Draw();
    }
}
```

This also needs a helper method to get the parent bone transform, so add this, too:

```
private Matrix GetParentTransform(Model m, ModelBone mb)
{
    return (mb == m.Root) ? mb.Transform :
        mb.Transform * GetParentTransform(m, mb.Parent);
}
```

So, what is going on here? You set the initial world matrix you want to use as the basis for everything, much like you did in the previous one-line call, but after that, things seem to get much more complicated! In reality, this is straightforward. For every mesh in your model, you go through the effects and set the matrices each needs. The view and projection matrices are the same for every effect (because you don't want the camera moving around based on which portion of the model you' render); however, the world matrix is vastly different.

Each mesh sets its world matrix to the parent's world matrix combined with the constant world matrix for the model. Notice that the helper method to get the parent's world matrix is recursive. Because each bone's transform is relative to its parent's transform, to get the full transform for any bone in the model, you need to combine its transform with all of its parents, which is what the helper method does. It stops when it gets to the root bone because it has no parent and simply returns the root bone's transform.

> **Note**
>
> In a real game, you do not want to use this recursive function every single frame as you did here to get the bone transforms. You would instead cache them (and use perhaps the `CopyAbsoluteBoneTransformsTo` helper method), but it was done this way to illustrate the concept.

If you want even more control, however, you can render everything in the model exclusively through the device methods without ever calling one of the helper `Draw` methods. Add yet another helper method to draw the model via the vertex data itself and update your `DrawModel` call to use this one instead:

```
private void DrawModelViaVertexBuffer(Model m, float radius, Matrix proj,
➥Matrix view)
{

Matrix world = Matrix.CreateScale(1.0f / radius);
    foreach (ModelMesh mm in model.Meshes)
    {
        foreach(ModelMeshPart mmp in mm.MeshParts)
        {
            IEffectMatrices iem = mmp.Effect as IEffectMatrices;
            if ( (mmp.Effect != null) && (iem != null) )
            {
                iem.World = GetParentTransform(m, mm.ParentBone) * world;
                iem.Projection = proj;
                iem.View = view;
                GraphicsDevice.SetVertexBuffer(mmp.VertexBuffer,
➥mmp.VertexOffset);
                GraphicsDevice.Indices = mmp.IndexBuffer;
                foreach (EffectPass ep in mmp.Effect.CurrentTechnique.Passes)
                {
                    ep.Apply();
                    GraphicsDevice.DrawIndexedPrimitives(
                        PrimitiveType.TriangleList, 0, 0,
                        mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
                }
            }
        }
    }
}
```

Notice the similarities between this one and the last one. You still need to enumerate through each of the meshes; however, instead of calling the `Draw` method on the meshes, enumerate through each of the mesh parts. You set the world, view, and projection matrices as you did before, although the actual rendering is done much differently. Much like you did in Chapter 4, the rendering happens directly through the `GraphicsDevice`. You learn in subsequent chapters that effects can have multiple passes (where you render the same thing multiple times with different effects), and this code handles the situation by enumerating through the effects before drawing.

You can see that the data needed to do the rendering is on the `ModelMeshPart` class. You need to set the vertex and index buffers to the device, which are on the part, and then the data needed to make the `DrawIndexedPrimitives` call are also there. By default, models loaded via the built-in importers from the content pipeline require the primitive type to be `TriangleList`, although creating your own importers can change this if you want to change it.

Summary

In this chapter, you learned about the camera and the view and projection matrices. You were introduced to the model class and learned how to render objects in the world with it.

In the next chapter, you look at the effects runtime and the built-in effects that are part of the framework.

This page intentionally left blank

6

Built-In Shader Effects

All modern games are rendered using a variety of shader effects, and the XNA Game Studio runtime comes with several built in effects. In this chapter you'll learn about them, including:

- BasicEffect
- The built in effect interfaces
- DualTextureEffect
- AlphaTestEffect
- EnvironmentMapEffect
- SkinnedEffect

You've actually used built-in effects for the rendering you've done up until now, even if you didn't realize it! In Chapter 4, "Introduction to 3D Graphics," you learned about pixel shaders and vertex shaders, which are combined into an effect. Although an effect doesn't have to have both a vertex shader and a pixel shader, you see the term effect used interchangeably throughout this book for the word shader. As mentioned before, a shader is a simple program that runs on the GPU (graphics hardware) that either manipulates vertices (the vertex shaders), pixels (the pixel shaders), or both.

XNA Game Studio 4.0 includes five built-in effects. `BasicEffect` has existed since the first version of XNA, and although the shader is quite complex, it provides the basic functionality you need to do common rendering. Version 4.0 includes four new built-in effect types, including `DualTextureEffect`, which supports rendering and blending two textures at once. The `AlphaTestEffect` is a simple shader for doing alpha tests. There is also the `EnvironmentMapEffect` that enables you to easily do environment maps (explained later), and the `SkinnedEffect` that enables basic character animations.

These five effects are guaranteed to work in any Reach profile game and work on all platforms. The same cannot be said for custom effects (which is discussed in Chapter 8, "Introduction to Custom Effects").

Using BasicEffect

Because it has been around the longest, it makes sense to start with `BasicEffect`. This effect is anything but basic, and the various permutations of things it can do are downright mind boggling at times. However, we might as well dive right in! Start by creating a new game project, and add box.fbx to the content project. Declare both the model variable and a basic effect:

```
Model model;
BasicEffect effect;
```

Update your `LoadContent` method to instantiate these objects:

```
model = Content.Load<Model>("box");
effect = new BasicEffect(GraphicsDevice);
effect.Projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
effect.View = Matrix.CreateLookAt(new Vector3(2, 3, 2), Vector3.Zero,
➥Vector3.Up);
```

Loading the model has been done several times, and creating the basic effect is pretty simple as well. It takes in the graphics device as its only parameter. You also use the `Projection` and `View` matrix parameters on the basic effect to set up your default camera. Because you're creating and manipulating your own effect (`BasicEffect`), add a `DrawModel` method that is similar to the previous chapter:

```
private Matrix GetParentTransform(Model m, ModelBone mb)
{
    return (mb == m.Root) ? mb.Transform :
        mb.Transform * GetParentTransform(m, mb.Parent);
}
private void DrawModel(Model m, Matrix world, BasicEffect be)
{
    foreach (ModelMesh mm in model.Meshes)
    {
        foreach (ModelMeshPart mmp in mm.MeshParts)
        {
            be.World = GetParentTransform(m, mm.ParentBone) * world;
            GraphicsDevice.SetVertexBuffer(mmp.VertexBuffer, mmp.VertexOffset);
            GraphicsDevice.Indices = mmp.IndexBuffer;
            be.CurrentTechnique.Passes[0].Apply();
            GraphicsDevice.DrawIndexedPrimitives(
                PrimitiveType.TriangleList, 0, 0,
                mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
        }
    }
}
```

This takes the model you passed in and renders it using the supplied world matrix and basic effect. It uses the mesh parts of the model to render directly, using the graphics device rather than the `Draw` helper methods on the model. `BasicEffect` has only one pass for its rendering, so the for loop around the effect apply is not needed.

> **Note**
>
> `BasicEffect` is the default effect that is created for this model during content importing, but you use a new one for ease of these examples.

Finally, replace your `Draw` method to render the model, and notice an ugly white box on the screen, much like you see in Figure 6.1.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    DrawModel(model, Matrix.Identity, effect);
}
```

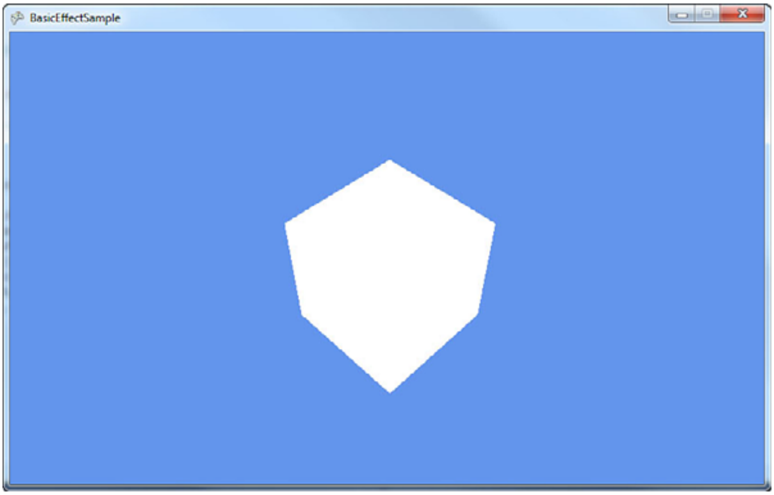


Figure 6.1 An ugly, unshaded box

As you can see, the box is not visually appealing. It doesn't even appear 3D! This is because it has no lighting on it, so everything has the same color, which means you can't see edges that make everything appear flat. Adding a simple call to `EnableDefaultLighting` on the effect after you have set the view matrix makes your box appear 3D because it turns on lights and makes the different faces of the box shaded differently. You've seen the box like this many times before. So instead, let's try something different.

Basic Lighting

If you added the call to `EnableDefaultLighting`, remove it and add the following after the view matrix is set in `LoadContent`:

```
effect.LightingEnabled = true;
```

This tells the effect to enable the lightning engine. Now run the application, and notice a similar shape as the first time you ran this application, but there is a bright white square in the center followed by the other portions of the block looking flat and black as in Figure 6.2.

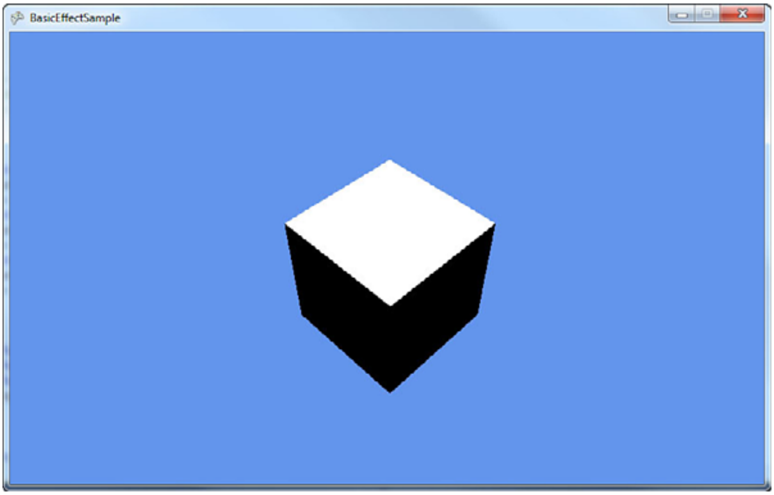


Figure 6.2 Default lighting

To understand how enabling lighting made portions of the box turn pitch black, first take a basic look at how lighting works. Basic effect gives you four different functions of lights for you to modify: an ambient light, directional lights, specular highlights, and the emissive color.

The ambient light is the easiest to visualize and understand. You can think of ambient light as the light that is always on, always around, comes from every direction, and lights everything equally. Notice an `AmbientLightColor` property on `BasicEffect` that defaults to black, which is why the sides of your box look pure black. Add the following line after enabling the lights in `LoadContent`:

```
effect.AmbientLightColor = Color.Purple.ToVector3();
```

Running the application now changes the black portions of the box to purple because that is the color of the ambient light. The top of the box is still bright white.

Note

Notice that all colors used in the built-in effects are not `Color` objects, but vector objects. The x, y, and z components of the vector object map to the red, green, and blue color components where 0.0f is fully off and 1.0f is fully on.`Vector3.Zero` is pure black, and `Vector3.One` is pure white. The `Color` object also has two helper methods, `ToVector3` and `ToVector4`, to easily get the appropriate data.

Changing the ambient light color to another color shows you that the portions of your box that were originally black and are now purple change to the color you are using as the ambient light color. If you change it back to white, you are back to where you started with a pure white square-like shape. That's because of the next light type, directional lights.

As the name implies, directional lights come from a certain direction. The sun is a good example of a directional light conceptually. The light goes in a particular direction infinitely without becoming more dim the farther away it goes, and without having a real source. Although the sun doesn't meet these criteria, from the perspective of the Earth, it is close enough. The built-in basic effect has support for up to three different directional lights, and by default, only the first one is enabled. These lights are accessed via three separate properties called `DirectionalLight0`, `DirectionalLight1`, and `DirectionalLight2`. Each directional light has a few properties that are interesting to look at, too.

First, the directional light has an `Enabled` property to determine whether it is on or not on. Much like the ambient light color, there is a `DiffuseColor` property that is the diffuse color of the light (you can think of this as simply the color of the light). This property defaults to pure white (Vector3.One), which is why the top of the box currently as white. A directional light also needs to have a `Direction` property that defaults to `Vector3.Down`, which also explains why the top of the box is pure white. The final property is the `SpecularColor`, which we discuss later in the chapter.

By switching `LightingEnabled` to true, you not only turn on lighting, but you also get the defaults for all of the lighting-centric properties, so a black ambient light and a single directional light turned on, pointing straight down with a white light. This explains the appearance of the box.

If you switch the direction, notice the box is lit differently. For example, add the following line after enabling the lights:

```
effect.DirectionalLight0.Direction = Vector3.Left;
```

This causes the top of the box to turn black again (the direction is no longer on top of it), but the right side of the box to turn white. This is because the direction of the light moves to the left, causing the right side to illuminate. If you use `Vector3.Right`, a pure black box appears because the left side of the box is lit, but you cannot see that side from where your camera is. Of course, having only one side of the cube light up isn't interesting, so change the direction slightly such as the following line:

```
effect.DirectionalLight0.Direction = Vector3.Normalize(new Vector3(-1,-1,0));
```

Both the top and the right portions of the box are now gray. Notice that you normalized the vector before passing it into the direction, because `Direction` is required to be a unit vector (meaning that the length of the vector must be one, which is also what `Normalize` does). The top and side are the same color gray because the angle between the direction of the light and the normal of the face is the same on each side with that vector (equally lighting the right and top faces). Change the direction to the following line:

```
effect.DirectionalLight0.Direction = Vector3.Normalize(new Vector3(-1,-1.5f,0));
```

All three sides of the cube now have different colors with the top as the lightest gray, the right as a dark gray, and the front as completely black.

Lighting Calculations

The final color of each piece of your model when it is lit comes from a wide variety of sources. Directional light is calculated using the dot product of the direction vector and the normal vector for the current face. The normal vector is perpendicular to the plane of the face. Without normal vectors, all lighting calculations (aside from ambient, which is everywhere equally) would not work. The color of the light is scaled by the dot product. That color is combined with vertex colors on the model, textures that are being rendered, and the ambient light.

If you change the color, the light changes to that color. Add the following line after setting the direction:

```
effect.DirectionalLight0.DiffuseColor = Color.Yellow.ToVector3();
```

The box is now a yellow color. If you use the previous direction, the top of the box is a light yellow, the right is a dark yellow, and the front is pitch black. This is because there is no light on the front of the box and it uses the ambient color (see Figure 6.3).

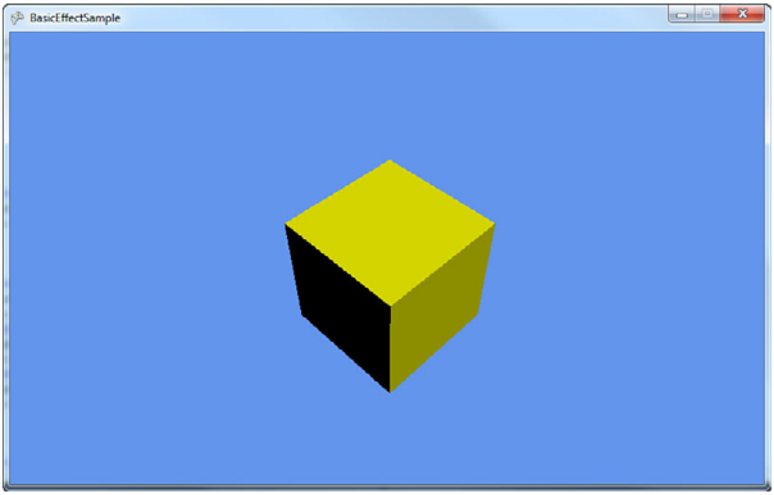


Figure 6.3 A box lit by a single directional light

The basic effect class supports up to three different directional lights. These lights can be turned on all at once, one at a time, or not at all. Just because they happen to be named in a numeric order doesn't mean you have to use them in that order. For example, add the following lines after your current light settings in `LoadContent`:

```
effect.DirectionalLight1.Enabled = true;
effect.DirectionalLight1.DiffuseColor = Color.Blue.ToVector3();
effect.DirectionalLight1.Direction = Vector3.Left;
```

This is setting the second of the directional lights basic effect uses to a blue color pointing to the left. Running the application shows you that the top square of the box is the same dark yellow color, and the right side (which now has a light yellow light as well as a solid blue light) is an odd purplish color. This is because the two light colors blend together. You can tell this by setting the `DirectionalLight0.Enabled` property to false and see that the right face is now a solid blue because it isn't combined with the yellow light. Set it back to true, so you can see the combination of all three lights. Replace your lightning code with the following lines:

```
effect.DirectionalLight0.Enabled = true;
effect.DirectionalLight0.DiffuseColor = Color.Red.ToVector3();
effect.DirectionalLight0.Direction = Vector3.Normalize(new Vector3(-1,-1.5f,0));
effect.DirectionalLight1.Enabled = true;
effect.DirectionalLight1.DiffuseColor = Color.Blue.ToVector3();
effect.DirectionalLight1.Direction = Vector3.Normalize(new Vector3(1, -1.5f, -1));
effect.DirectionalLight2.Enabled = true;
effect.DirectionalLight2.DiffuseColor = Color.Green.ToVector3();
effect.DirectionalLight2.Direction = Vector3.Normalize(new Vector3(0, -1.5f, -1));
```

This takes three different lights: red, green, blue, shining all on the box in different directions. Each light hits more than one face, so you see a combination of the colors (mostly red on the right face, a pinkish color on the top face, and mostly blue on the front face). Each of these lights can be controlled individually or combined.

There is also a helper method on `BasicEffect` called `EnableDefaultLighting`, which sets up a standard lighting rig. This is actually not something unique to games or even 3D graphics. Photographers and movie makers discovered long ago that taking pictures of things looked much better when using more than one light and specifically, they looked best with at least three lights (this is also why this class supports three lights).

This is normally done with a key light that is the brightest of the lights and is pointed directly at the subject. A fill light that is dimmer also points at the subject, although usually at a ninety degree angle from the key light. This light helps balance out the image by illuminating some of the shadows that are too dark from the key light. Finally, the backlight, which as the name implies, shines from behind the subject to help separate him or her from the background and to give subtle detail. The back light is normally the dimmest light.

With all the basics of directional lighting out of the way, let's move on to the other major light, specular lighting. This is used in the standard lighting rig that the `EnableDefaultLighting` helper method creates for you. What are specular highlights?

A specular highlight is the bright spot on an object. It makes objects appear shiny. You can't see this on a box though, so let's make a few changes to the current project. First, delete the lighting code you added so far (you should just have the world and view matrices set on the effect). Next, add the `sphere.x` object from the downloadable examples to your content project. It is much easier to see this effect on a sphere. Finally, change your model loading to use the sphere rather than the box.

Run the application and notice a white circle in the middle of your screen, much like you saw the box before. Again, this is because there are no lights in the scene anymore. So first, turn on a directional light, so you can see the sphere as a sphere, not a circle, as follows:

```
effect.LightingEnabled = true;
effect.DirectionalLight0.Enabled = true;
effect.DirectionalLight0.DiffuseColor = Color.Red.ToVector3();
effect.DirectionalLight0.Direction = Vector3.Normalize(new Vector3(0, -1.5f, -1));
```

The sphere looks like a sphere rather than a circle, but it doesn't have a shiny look. This is because the specular color of the directional light defaults to black, so you do not see it. Add the following line directly after setting the direction:

```
effect.DirectionalLight0.SpecularColor = Color.White.ToVector3();
```

This turns on the specular color to a full white color, and much like you see in Figure 6.4, your sphere now has a bright white spot on it.

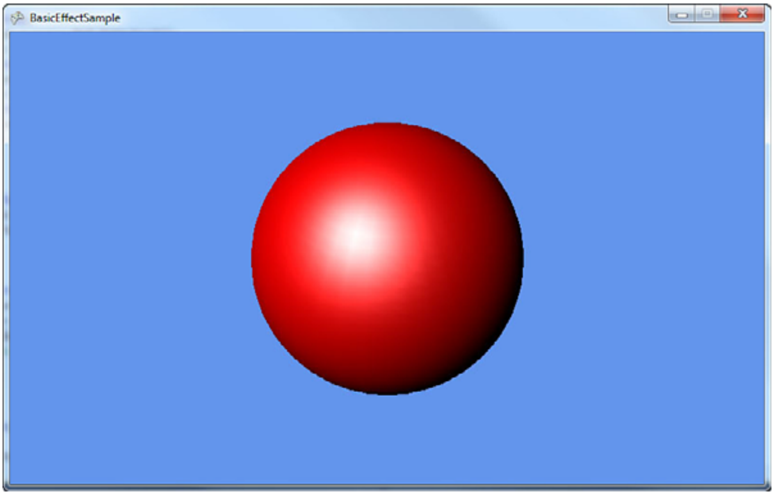


Figure 6.4 A sphere with a specular highlight

The specular color is much like the other colors because you can combine it with any of the others. The white specular light is combined with the red diffuse light and forms the bright spot. If you change the specular color from white to blue, notice that your white spot turns more purplish (the red diffuse light and the blue specular light combine to a purple color). Each directional light in the effect includes a specular color.

There are two other properties of specular lighting that are on the effect itself, not on any of the directional lights:the `SpecularColor` of the material and the `SpecularPower` of the effect. The power property is a float value that determines how concentrated the shiny spot is. Higher values make the effect more concentrated (a smaller spotlight on the sphere), and larger values lower the concentration and make the spotlight effect larger.

To see this behavior in action, add the following code to your `Draw` method before the `DrawModel` call:

```
effect.SpecularPower = (float)gameTime.TotalGameTime.TotalSeconds * 2.0f;
```

This starts the specular power at 0 (and your sphere looks almost completely white) and raises it higher as the application runs (it gains a power of 1.0f every half second). Notice that the spotlight effect quickly concentrates from the full sphere to a smaller spot, and then slowly gets more concentrated to a small point if you let it run long enough. Remove the line and let's talk about the specular material color.

The `SpecularColor` on the effect is the color that is reflected by the lights. The default of this property is white, so all color is reflected making the spotlight effect white. Add this line after setting the light's specular color in the `LoadContent` method:

```
effect.SpecularColor = Color.Red.ToVector3();
```

This reflects only the red light of the white specular light. If you run this, the sphere looks similar to before, but rather than a bright white spot, you see a red spot instead. If you change the `SpecularColor` on the directional light to a color without red light, for example, say blue, the specular effect completely disappears. This is because there is no red light to reflect a pure blue light and no specular effect.

Notice that you can see portions of how the sphere was created in the light because the vertices of the sphere stick out in the spotlight effect. This is because colors are determined by default per vertex, which show you these vertices when they're close. If your graphics card can support it, you can have this effect do its lighting calculations per pixel rather than per vertex to give a much more smooth (and realistic) effect.

Add this line to the end of your lighting code in `LoadContent` to turn on per pixel light if your system supports it:

```
effect.PreferPerPixelLighting = true;
```

Notice that the spotlight is smooth. See Figure 6.5 for a comparison between per vertex and per pixel lighting.

> **Note**
>
> Per pixel lighting can look amazing, but at a cost. On Windows Phone 7, this process can hinder performance. This is discussed more in depth in Chapter 11, "Understanding Performance."

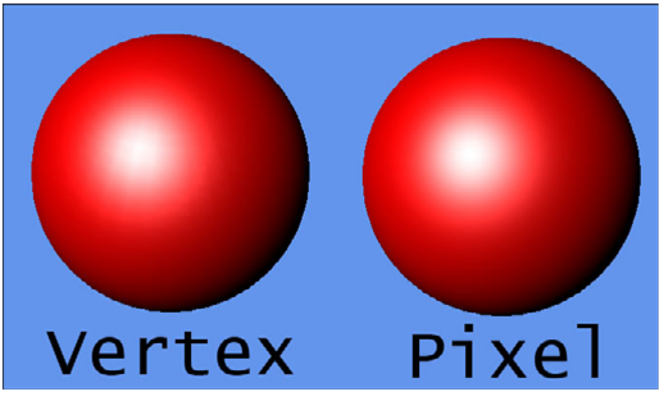


Figure 6.5 Per-pixel and per-vertex comparison

The last lighting property is the `EmissiveColor` of the material. Unlike the specular material, this color is not reflected. It can be thought of as the color of light that the object actually emits. The default value of this is a pure black color (the object doesn't emit anything). To see how this affects your sphere, replace your lighting code in `LoadContent` (after view and projection matrices are set) with the following:

```
effect.LightingEnabled = true;
effect.DirectionalLight0.Enabled = true;
effect.DirectionalLight0.DiffuseColor = Color.White.ToVector3();
effect.DirectionalLight0.Direction = Vector3.Normalize(new Vector3(-1, -1.5f, 0));
```

This makes a sphere lit with a single white directional light. To make the sphere emit some red light, add the following line:

```
effect.EmissiveColor = Color.Red.ToVector3();
```

Notice a vastly different sphere. The dark portions without the emissive color are now bright red, and they lighten to white as the light from the directional light takes over. If you switch the color of the directional light to blue, it lightens to purple instead of white, because the blue light is combining with the red light to form purple.

Textures, Vertex Colors, and Fog

The basic effect also handles rendering a texture. If the model you are loading from the content pipeline includes textures, they are included as part of your model and rendered automatically. If they aren't included, you can control that directly through the effect. To do this, add a new texture variable:

```
Texture2D texture;
```

To include a texture to load, add the file `cat.jpg` from your content project. Next, in your `LoadContent` method, switch the model you're loading back to the box you used earlier. Remove the lighting code you used for the emissive color example, and create the texture and set it to the effect, as follows:

```
texture = Content.Load<Texture2D>("cat");
effect.TextureEnabled = true;
effect.Texture = texture;
```

Running the application shows you the box again, except this time you see each face of the box covered in the cat (see Figure 6.6).

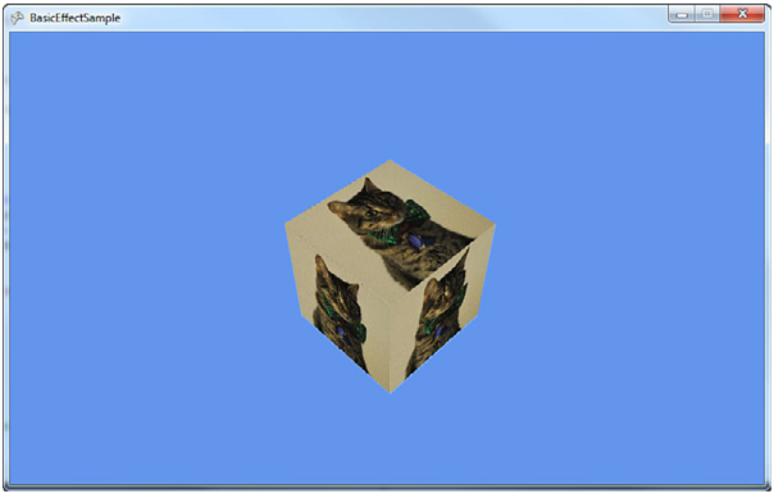


Figure 6.6 A textured box

Notice that each face of the box looks the same. Because you're using `BasicEffect`, you can use the other features as well. Add the standard lighting rig in your `LoadContent` method:

```
effect.EnableDefaultLighting();
```

The top of box is now slightly lighter than the other two faces due to the lighting. If you had a different model, you would see the specular highlights on it. You don't see them because you rendered a cube, which has only a small number of vertices.

You're almost done with the `BasicEffect` class. You can also use this effect for rendering vertex colors that are baked into the model. First, delete the code in your `LoadContent` method after the view and projection matrices are set, and add the following line:

```
effect.VertexColorEnabled = true;
```

This turns on the mode to render with vertex colors enabled. If you run the application, you see a black square-like blob. This is because the box model has no vertex colors in it. Let's modify the model to add vertex colors. Add a new vertex buffer variable that will hold the updated data:

```
VertexBuffer vb;
```

Before you create the vertex buffer to hold the updated data, define the structure for what kind of data it will hold. The structure does not exist, so you need to create it. Add the following lines to your project:

```
struct VertexColor : IVertexType
{
    public Vector3 Position;
    public Vector3 Normal;
    public Vector2 UV;
    public Color Color;
    public static readonly VertexDeclaration _decl = new VertexDeclaration(
        new VertexElement(0, VertexElementFormat.Vector3,
            VertexElementUsage.Position, 0),
        new VertexElement(12, VertexElementFormat.Vector3,
            VertexElementUsage.Normal, 0),
        new VertexElement(24, VertexElementFormat.Vector2,
            VertexElementUsage.TextureCoordinate, 0),
        new VertexElement(32, VertexElementFormat.Color,
            VertexElementUsage.Color, 0));
    public VertexColor(VertexPositionNormalTexture data)
    {
        Position = data.Position;
        Normal = data.Normal;
        UV = data.TextureCoordinate;
        Color = Color.White;
    }
    public VertexDeclaration VertexDeclaration
    {
        get { return _decl; }
    }
}
```

Whew, that's a lot of information, so let's break it down. First, notice that the structure expects to implement `IVertexType`. This is used to describe the data to the graphics device, and has only one method to get the `VertexDeclaration`. Next, you have the actual data you want to store. The current box has three pieces of data it stores: the position of each vertex, the normal of each vertex, and the texture coordinates of each vertex. You keep these three, but you also add the color.

Next, you create a static `VertexDeclaration` (so you don't create a new one every time you use the property that the interface implementation requires). This is done by describing the data this structure has in a language the graphics device understands. This is done through an array of `VertexElement`, with each of the members in the array describing a piece of the data that will be held. Let's look at the `VertexElement` constructor that's used here.

The first parameter of the constructor is the offset of the data (in bytes) from the beginning of the vertex. For the first member, use zero; for the second member, use 12. This is because the first piece of data is a `Vector3`, which contains three floats that are each four bytes large, and three times four is 12. The others are calculated in the same manner. The second parameter in the constructor is the format of the data. In your first two members, your data is in the `Vector3` format. Your third member is a `Vector2`, and the last is a `Color`. This parameter tells the graphics device how much data is required for each member.

The third parameter of the constructor is how the effects will use this data. In the first member, it is a `Position`, the second is a `Normal`, the third is a `TextureCoordinate`, and the last is a `Color`. You can ignore the last parameter for now—just use zero.

There is then a constructor on the structure itself taking in the data type of vertices in the model, `VertexPositionNormalTexture`, which uses the old data. This constructor simply sets the default color to white, and then uses the data from the passed in structure to fill out the current structure.

This describes the new data, so let's take a look at creating the vertex buffer and filling it up with good data. Add the following lines to the end of your `LoadContent` method:

```
int numberVertices = model.Meshes[0].MeshParts[0].NumVertices;
vb = new VertexBuffer(GraphicsDevice, typeof(VertexColor),
    numberVertices, BufferUsage.None);
VertexPositionNormalTexture[] data =
new VertexPositionNormalTexture[numberVertices];
VertexColor[] newData = new VertexColor[numberVertices];
model.Meshes[0].MeshParts[0].VertexBuffer.GetData
➥<VertexPositionNormalTexture>(data);
// Copy the data
for (int i = 0; i < numberVertices; i++)
{
    newData[i] = new VertexColor(data[i]);
    newData[i].Color = Color.Red;
}
vb.SetData<VertexColor>(newData);
```

The box model you used has exactly one mesh with one mesh part in it, so getting the total number of vertices here is easy. You have to do a bit more work to make this more generic, which is beyond the scope of this example. You can then create your new vertex buffer using the type of structure you previously defined that dictates what type of data you're using.

Now you need to get the old data from the model. First, create two new arrays of vertices: one of the type `VertexPositionNormalTexture`, which is the type of data currently in the model, and the other of type `VertexColor`, the new data you use. Then, you can simply call the `GetData` method passing in the first array to have the model's data push into that array. You then can use the for loop to go through each of pieces of old data, filling up the new data and setting the vertex color to red. Finally, call `SetData` on your vertex buffer to fill it with the newly created data including the vertex color.

You're not quite ready to render yet because your current `DrawModel` call uses the model's vertex buffer, not the one you just created. Add the following method to draw with a specific vertex buffer:

```
private void DrawModelWithVertexBuffer(Model m, VertexBuffer v,
    Matrix world, BasicEffect be)
{
    foreach (ModelMesh mm in model.Meshes)
    {
        foreach (ModelMeshPart mmp in mm.MeshParts)
        {
            be.World = GetParentTransform(m, mm.ParentBone) * world;
            GraphicsDevice.SetVertexBuffer(v, 0);
            GraphicsDevice.Indices = mmp.IndexBuffer;
            be.CurrentTechnique.Passes[0].Apply();
            GraphicsDevice.DrawIndexedPrimitives(
                PrimitiveType.TriangleList, 0, 0,
                mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
        }
    }
}
```

This is almost identical to the previous `DrawModel` call; it uses the passed in vertex buffer instead. Update your `Draw` method to call this instead of `DrawModel`, and notice the black box is now a red box. However, just like everything else, this effect can be combined with all the other effects. For example, to see the box shaded by the standard lighting rig, add the following line after your `SetData` call in `LoadContent`:

```
effect.EnableDefaultLighting();
```

You can texture this as well by adding the following code:

```
texture = Content.Load<Texture2D>("cat");
effect.Texture = texture;
effect.TextureEnabled = true;
```

Now, the box with the cat picture has everything shaded red. If you change the previous vertex color to blue, everything is shaded blue.

The last thing to talk about is the fog effect. Your project is a bit crowded, so to show this, replace your `LoadContent` method with the following lines:

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    model = Content.Load<Model>("sphere");
    effect = new BasicEffect(GraphicsDevice);
    effect.Projection = Matrix.CreatePerspectiveFieldOfView(
       MathHelper.PiOver4, GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
    effect.View = Matrix.CreateLookAt(
       new Vector3(0, 8, 22), Vector3.Zero, Vector3.Up);
    effect.EnableDefaultLighting();
    effect.SpecularColor = Vector3.Zero;
    effect.PreferPerPixelLighting = true;
    effect.FogColor = Color.Red.ToVector3();
    effect.FogEnabled = true;
    effect.FogStart = 10.0f;
    effect.FogEnd = 40.0f;
}
```

The majority of the previous code should be old hat by now. You switched back to the sphere to see the effect slightly better. You modified the view matrix to be a little bit higher and quite a bit farther away, and then turned on the standard lighting rig. Setting the specular color back to black isn't necessary, but it's easier to see the effect without the specular highlights.

Next, let's set the four fog parameters. Enabling the fog is pretty straightforward, as is setting the color. The start and end parameters determine where the fog effect starts and where it ends, although "end" is a bit of a misnomer. These numbers are distances from the camera, where everything closer than the start value has no fog effect, everything beyond the end value has the full fog effect, and everything in between those values scale linearly between no effect and full effect.

It is easier to visualize this by seeing it in action, so let's update the `Draw` method to render some spheres. Delete everything from this method except the `Clear` call and add the following lines:

```
const int NumberSpheres = 55;
for (int i = 0; i < NumberSpheres; i++)
{
    Matrix world =Matrix.CreateTranslation(
        ((i % 5) - 2) * 4, 0, (i/5-3) * -4);
    DrawModel(model, world, effect);
}
```

This renders a set of fifty-five spheres across the screen (you can change the constant to lower/raise the amount you are rendering). The spheres start close to the screen, and every set of five moves the next set a bit farther from the camera. Eventually, as you get ten units away, the spheres start having a red tint. The farther way they get from the camera, the moreof the tint they get until they're completely red (see Figure 6.7).

Fog adds another way to set the final color of objects in the world. By using fog, you can make objects appear to "disappear" the farther away they are. Using red doesn't make that apparent, but go back and change the `FogColor` in your `LoadContent` method to `CornflowerBlue`. Much like you see in Figure 6.8, the farthest spheres simply disappear, while the ones in the fog range appear to be faded.

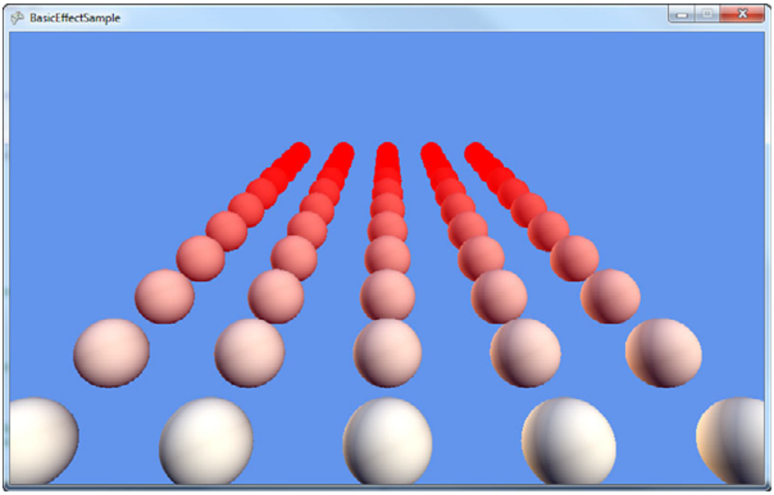

BasicEffectSample

Figure 6.7 A first look at fog

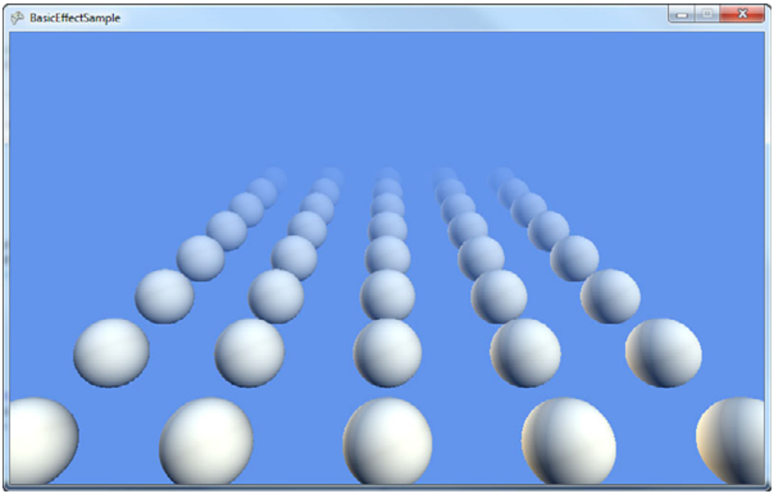

BasicEffectSample

Figure 6.8 Spheres disappearing in fog

As before, all of these effects that `BasicEffect` uses can be combined with each other. To show this, switch your model back to the box, and add the following to the end of the `LoadContent` method:

```
texture = Content.Load<Texture2D>("cat");
effect.TextureEnabled = true;
effect.Texture = texture;
```

Notice your spheres are replaced with boxes that have pictures of cats on them, and they slowly fade out into the distance (see Figure 6.9).

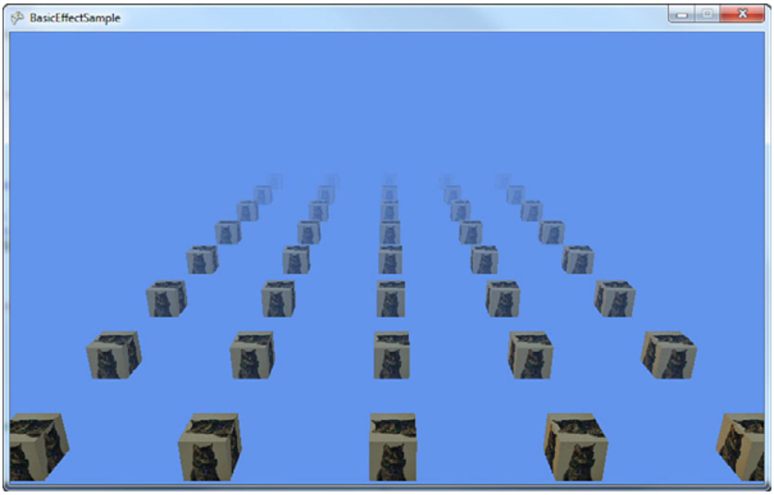


Figure 6.9 Textured boxes disappearing in fog

As you can see, `BasicEffect` does quite a bit. Before moving on to the other built-in effects, let's take a few minutes to see the interfaces it uses.

Using the Effect Interfaces

`BasicEffect` implements three different interfaces that are generic enough to be used by multiple types of effects. Each of the built-in effects in Game Studio 4.0 uses at least one of these interfaces, and your own custom effects could also. This enables you to easily cast your effects to these interface types to do common operations. The most common is the following:

```
public interface IEffectMatrices
{
    Matrix Projection { get; set; }
    Matrix View { get; set; }
    Matrix World { get; set; }
}
```

This is to give you a spot to set your matrices generically. As expected, there is also one to handle basic lighting:

```
public interface IEffectLights
{
    Vector3 AmbientLightColor { get; set; }
    DirectionalLight DirectionalLight0 { get; }
    DirectionalLight DirectionalLight1 { get; }
    DirectionalLight DirectionalLight2 { get; }
    bool LightingEnabled { get; set; }
    void EnableDefaultLighting();
}
```

Finally, there is an interface you can use if your effect supports fog:

```
public interface IEffectFog
{
    Vector3 FogColor { get; set; }
    bool FogEnabled { get; set; }
    float FogEnd { get; set; }
    float FogStart { get; set; }
}
```

All of these interfaces are used by the built-in effects and are there for you to implement in your own custom effect classes to write generic code for many common operations. Now let's move on to the next built-in effect type.

Using DualTextureEffect

You might have noticed that while using `BasicEffect`, you can use only a single texture. The `DualTextureEffect` enables you to use two. A common use of a dual texture is for either light maps or decals. This effect implements `IEffectMatrices` and `IEffectFog`, but not `IEffectLights`, so you don't get lights here! Because a common use of this is for light maps, that makes sense, too. Let's see it in action. Create a new project and add a few files in your content project. Add the model (dualtextureplane.fbx), the first texture (ground.jpg), and the light map (lightmap.png). Also add a couple new variables to your project:

```
Model model;
DualTextureEffect effect;
```

To show this effect, render a single plane with a texture on it (it can be the "ground"). To do this, add the following code to your `LoadContent` method:

```
model = Content.Load<Model>("dualtextureplane");
effect = new DualTextureEffect(GraphicsDevice);
effect.Projection = Matrix.CreatePerspectiveFieldOfView(
    MathHelper.PiOver4, GraphicsDevice.Viewport.AspectRatio, 1.0f, 50.0f);
effect.View = Matrix.CreateLookAt(
    new Vector3(0, 32, 32), Vector3.Zero, Vector3.Up);
effect.DiffuseColor = Color.DarkGray.ToVector3();
effect.Texture = Content.Load<Texture2D>("ground");
```

This loads your model and effect and sets up your camera. It also sets the first texture and updates the diffuse color (the default is bright white). You need to add a `DrawModel` call to use this effect, so add that as follows:

```
private void DrawModel(Model m, Matrix world, DualTextureEffect be)
{
    foreach (ModelMesh mm in m.Meshes)
    {
        foreach (ModelMeshPart mmp in mm.MeshParts)
        {
            be.World = world;
            GraphicsDevice.SetVertexBuffer(mmp.VertexBuffer,mmp.VertexOffset);
            GraphicsDevice.Indices = mmp.IndexBuffer;
            be.CurrentTechnique.Passes[0].Apply();
            GraphicsDevice.DrawIndexedPrimitives(
                PrimitiveType.TriangleList, 0, 0,
                mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
        }
    }
}
```

This is similar to your last `DrawModel` call, except without the generalization of enabling a hierarchy of world transforms, it simply sets the effect and draws the geometry. Now include a call to this method in your `Draw` call:

```
DrawModel(model, Matrix.Identity, effect);
```

After you run this, you expect to see the plane rendered on the screen with the single texture showing up, but all you see is a black square. This is because you don't have the second texture loaded, and the default color it returns if the texture isn't there is black. It then combines the two colors from the textures, but black combined with anything else gives you black, so you see nothing. Temporarily set your second texture to light grey using the following code at the end of your `LoadContent` method:

```
effect.Texture2 = new Texture2D(GraphicsDevice, 1, 1);
effect.Texture2.SetData(new Color[] { new Color(196, 196, 196, 255) });
```

This shows you the plane rendered on the ground with a single texture. Well, it is two textures, with the second texture as a single light gray pixel. Not hugely exciting, but if you add this line to the end of your `LoadContent` method, the second texture is loaded, and running the application makes the ground appear to have spotlights on it (see Figure 6.10):

```
effect.Texture2 = Content.Load<Texture2D>("lightmap");
```

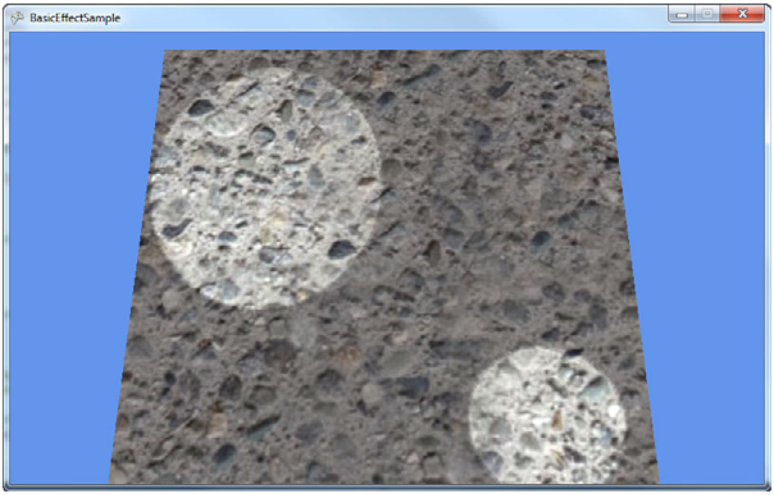


Figure 6.10 Spotlights with DualTextureEffect

You can also set `effect.Texture` to the single light gray pixel and see the light map rendered alone on the plane. With this effect, you can perform quite a range of techniques using two textures, and light maps as one of them!

Using AlphaTestEffect

At a high level, the `AlphaTestEffect` enables you to get performance benefits by quickly (and early) rejecting pixels based off their alpha level. There are only a few things important on this effect, so we won't have a large example here.

Like the `DualTextureEffect`, `AlphaTestEffect` implements the `IEffectMatrices` and `IEffectFog` interfaces. There are two important properties on this effect: first is the `AlphaFunction`, which describes how to compare the alpha, and second is the `ReferenceAlpha`, which is what each pixel's alpha component is compared to (using the `AlphaFunction`).

The `CompareFunction` enumeration has a variety of ways each pixel can be compared to the `ReferenceAlpha`, and if the compare function evaluates as true, that pixel is accepted, processed, and potentially rendered. If you use the `Always` value, all pixels are accepted much like the `Never` value accepts no pixels. Each of the others behaves as expected with `Less`, `LessEqual`, `Equal`, `Greater`, `GreaterEqual`, and `NotEqual`.

Using EnvironmentMapEffect

Have you ever seen (either in the real world or in a game) a brand new car sitting outside on a bright and sunny day? Did you notice how the car looked so shiny? Did you notice how the car reflected the outside world on itself? This effect is called the

`EnvironmentMapEffect` because you map the environment onto an object. Like `BasicEffect`, this effect implements all three interfaces.

Environment maps are done by rendering a cube texture (the `TextureCube` class) as a reflection over any objects that need to have the reflection. A *cube texture* is (as the name implies) a texture that covers a cube, so it has six different faces. However, it covers the inside of the cube. The effect is rendered as if the object is sitting inside the cube with the texture reflecting off the object.

To show this effect, create a new game project. Add some content from the downloadable examples to your content project. You need the model that you will render the reflection, so add the `environmentmapmodel.fbx` model. Add the six textures the skybox will use (skybox0-5.jpg).

Next, add a few variables:

```
Model model;
EnvironmentMapEffect envEffect;
TextureCube texture;
```

Like always, you'll need to instantiate these variables, too, so update your `LoadContent` method by adding the following:

```
model = Content.Load<Model>("environmentmapmodel");
envEffect = new EnvironmentMapEffect(GraphicsDevice);
envEffect.Projection = Matrix.CreatePerspectiveFieldOfView(
    MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
envEffect.View = Matrix.CreateLookAt(
    new Vector3(2, 3, 32), Vector3.Zero, Vector3.Up);
texture = new TextureCube(GraphicsDevice, 256,
    false, SurfaceFormat.Color);
Color[] facedata = new Color[256 * 256];for (int i = 0; i < 6; i++)
{
    Texture2D tempTexture = Content.Load<Texture2D>(
    "skybox" + i.ToString());
    tempTexture. GetData<Color>(facedata);
    texture.SetData<Color>((CubeMapFace)i, facedata);
}
envEffect.Texture = (model.Meshes[0].Effects[0] as BasicEffect).Texture;
envEffect.EnvironmentMap = texture;
envEffect.EnableDefaultLighting();
envEffect.EnvironmentMapAmount = 1.0f;
envEffect.FresnelFactor = 1.0f;
envEffect.EnvironmentMapSpecular = Vector3.Zero;
```

The beginning of this code is familiar: Load the model, create the effect, and set up the camera. You then need to create a new `TextureCube` object that is used as your environment map, and that needs to be filled with data.

Note

In a real game, you probably wouldn't want to load your cube texture (environment map) in this manner; you would use a custom content pipeline processor. The Content Pipeline will be discussed in depth in Chapter 9, "Using the Content Pipeline."

To easily fill the environment map with data, load each face of the cube map into its own temporary texture and copy the data to the appropriate face of the cube texture. After the cube texture has data, set the parameters for the effect and you're ready to start rendering.

The `Texture` parameter, much like `BasicEffect`, is simply the texture the object is supposed to have, what it would look like without reflections. The `EnvironmentMap` parameter is the cube texture that contains the reflection data, and `EnvironmentMapAmount` is a float describing how much of the environment map could show up (from 0.0f, or nothing, to 1.0f, or the max). The default amount is 1.0f.

The `FresnelFactor` parameter describes how Fresnel lighting should be applied to environment map, independent of the viewing angle. With a low value, the environment map is visible everywhere. With a value of 0.0f, the environment map covers the entire object, and a large value shows only the environment around the edges. The default value is also 1.0f.

The `EnvironmentMapSpecular` parameter is a bit special. It blends the alpha from the environment map into the final image using this value. This enables you to embed maps into your environment map to have cheap specular highlights (rather than having to actually use the GPU to render more lights). Even though it is set, its default value is also `Vector3.Zero`.

Lastly, enable the standard lighting rig. Now you're ready to render this model. Include the `DrawModel` method you used several times so far.

```
private void DrawModel(Model m, Matrix world, EnvironmentMapEffect be)
{
    foreach (ModelMesh mm in m.Meshes)
    {
        foreach (ModelMeshPart mmp in mm.MeshParts)
        {
            be.World = world;
            GraphicsDevice.SetVertexBuffer(mmp.VertexBuffer,
                 mmp.VertexOffset);
            GraphicsDevice.Indices = mmp.IndexBuffer;
            be.CurrentTechnique.Passes[0].Apply();
            GraphicsDevice.DrawIndexedPrimitives(
                PrimitiveType.TriangleList, 0, 0,
                mmp.NumVertices, mmp.StartIndex, mmp.PrimitiveCount);
        }
    }
}
```

Finally, to see how the model reflects the environment map, add the following lines to the `Draw` method and see your reflection map (see Figure 6.11).

```
float time = (float)gameTime.TotalGameTime.TotalSeconds;
DrawModel(model, Matrix.CreateRotationY(time * 0.3f) *
    Matrix.CreateRotationX(time), envEffect);
```

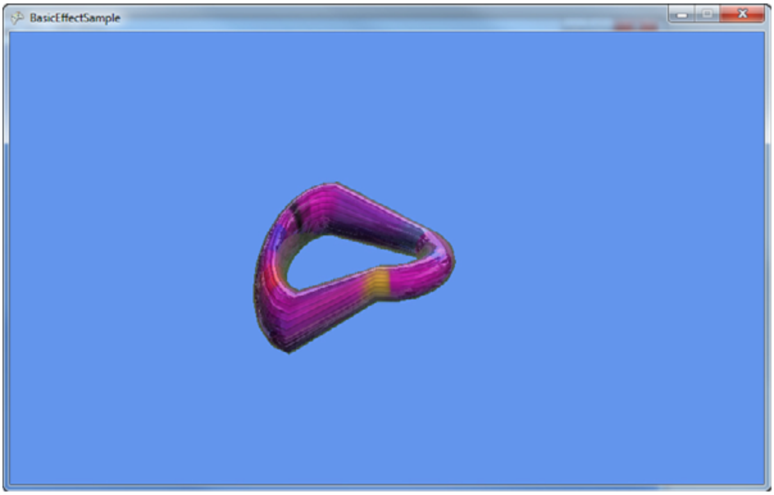


Figure 6.11 An environment map

As the model rotates, you can see the reflection of the environment around it. Manipulating the `EnvironmentMapAmount` parameter as well as the `FresnelFactor` parameter changes how much reflection you see. With that, let's move on to the last of the built-in effects.

Using SkinnedEffect

Much like `BasicEffect` and `EnvironmentMapEffect`, `SkinnedEffect` implements all three of the standard interfaces: `IEffectMatrices`, `IEffectLighting`, and `IEffectFog`. This section focuses only on the new pieces of functionality. This effect is primarily used to enable you to render animated objects (or "skinned" objects).

Animation for a particular model is normally stored within the model itself and is included when you are importing and processing the model. However, by default, you won't be able to get this data at runtime, so you need to build a content processor to store this data.

> **Note**
>
> Content importers and processors are discussed more in depth in Chapter 9—think of this as a crash course for a processor.

This appears to be the most complex example you've seen so far, but despite the number of projects, it isn't that complicated. First, you need the actual example project, so create a new game project called `SkinnedEffectExample`.You also need two other projects though: your content processor project and another game library project to hold these shared types.

In your solution, you should have two projects: the `SkinnedEffectExample` game project and the `SkinnedEffectExampleContent` project. Right-click your solution file in Visual Studio, select Add -> New Project, and then choose a Game Library project that will hold all of the shared types between your game and the content processor.You can call this one `SkinningInformation`. Lastly, right-click the solution again in Visual Studio and choose Add -> New Project, but this time choose the Content Pipeline Extension Library.You can call this `SkinnedModelProcessor`.

To make sure that both your game and the content processor have access to your shared data types, right-click the References node (located under your `SkinnedModelProcessor` project) and select Add Reference. Choose the Projects tab, and select your `SkinningInformation` project. Next, do the same thing for the `SkinnedEffectExample` project, and now you're basically ready to get started.

To add the shared data types, remove the class1.cs from your `SkiningInformation` project, right-click the project, select Add -> New Item, and choose a new code file called `SharedTypes.cs`. Add the following code to that file:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
 using Microsoft.Xna.Framework.Content;
public class Keyframe
{
    public Keyframe(int bone, TimeSpan time, Matrix transform)
    {
        Bone = bone;
        Time = time;
        Transform = transform;
    }
    private Keyframe() { }
    [ContentSerializer]
    public int Bone { get; private set; }
    [ContentSerializer]
    public TimeSpan Time { get; private set; }
    [ContentSerializer]
    public Matrix Transform { get; private set; }
}
```

The first shared type is a `KeyFrame`. This is essentially the state of a bone at a given point in time. Remember that the bone describes the transform of a particular portion of the model. By using key frames, a model can define a wide variety of animations by simply having the bones placed at particular places in time.

Next, you need another piece of shared data, so add the following lines to the code file:

```
public class AnimationClip
{
    public AnimationClip(TimeSpan duration, List<Keyframe> keyframes)
    {
        Duration = duration;
        Keyframes = keyframes;
    }
    private AnimationClip() { }
    [ContentSerializer]
    public TimeSpan Duration { get; private set; }
    [ContentSerializer]
    public List<Keyframe> Keyframes { get; private set; }
}
```

This defines the total animation as a list of key frames. The duration is the total length of animation, and the key frames are also stored. Because this class is self-explanatory, add another shared type here:

```
public class SkinningData
{
    public SkinningData(Dictionary<string, AnimationClip> animationClips,
                        List<Matrix> bindPose, List<Matrix> inverseBindPose,
                        List<int> skeletonHierarchy)
    {
        AnimationClips = animationClips;
        BindPose = bindPose;
        InverseBindPose = inverseBindPose;
        SkeletonHierarchy = skeletonHierarchy;
    }
    private SkinningData() { }
    [ContentSerializer]
    public Dictionary<string, AnimationClip> AnimationClips { get;
         private set; }
    [ContentSerializer]
    public List<Matrix> BindPose { get; private set; }
    [ContentSerializer]
    public List<Matrix> InverseBindPose { get; private set; }
    [ContentSerializer]
    public List<int> SkeletonHierarchy { get; private set; }
}
```

This is the data type that holds the animations for use at runtime. It has a few properties, namely a list of animation clips that the model supports along with the bind pose, the inverse of the bind pose, and a skeleton hierarchy. The bind pose is the "pose" of the model before any animation occurs (where the inverse is, of course, the inverse of this). The skeleton hierarchy is the list of indices into the bones.

Notice that many of the properties in these shared data types include the `ContentSerializer` attribute. This is discussed later in Chapter 9.

There is one more type to add before you can start with the processor. Add the following class:

```
public class AnimationPlayer
{
    AnimationClip currentClipValue;
    TimeSpan currentTimeValue;
    int currentKeyframe;
    Matrix[] boneTransforms;
    Matrix[] worldTransforms;
    Matrix[] skinTransforms;
    SkinningData skinningDataValue;
    public AnimationPlayer(SkinningData skinningData)
    {
        if (skinningData == null)
            throw new ArgumentNullException("skinningData");
        skinningDataValue = skinningData;
        boneTransforms = new Matrix[skinningData.BindPose.Count];
        worldTransforms = new Matrix[skinningData.BindPose.Count];
        skinTransforms = new Matrix[skinningData.BindPose.Count];
    }
    public void StartClip(AnimationClip clip)
    {
        if (clip == null)
            throw new ArgumentNullException("clip");
        currentClipValue = clip;
        currentTimeValue = TimeSpan.Zero;
        currentKeyframe = 0;
        // Initialize bone transforms to the bind pose.
        skinningDataValue.BindPose.CopyTo(boneTransforms, 0);
    }
    public void Update(TimeSpan time, bool relativeToCurrentTime,
                       Matrix rootTransform)
    {
        UpdateBoneTransforms(time, relativeToCurrentTime);
        UpdateWorldTransforms(rootTransform);
        UpdateSkinTransforms();
    }
    public void UpdateBoneTransforms(TimeSpan time,
        bool relativeToCurrentTime)
    {
        if (currentClipValue == null)
            throw new InvalidOperationException(
                        "AnimationPlayer.Update was called before
➥StartClip");
        // Update the animation position.
        if (relativeToCurrentTime)
        {
            time += currentTimeValue;
            // If we reached the end, loop back to the start.
            while (time >= currentClipValue.Duration)
                time -= currentClipValue.Duration;
        }
        if ((time < TimeSpan.Zero) || (time >= currentClipValue.Duration))
            throw new ArgumentOutOfRangeException("time");
        // If the position moved backwards, reset the keyframe index.
        if (time < currentTimeValue)
        {
            currentKeyframe = 0;
            skinningDataValue.BindPose.CopyTo(boneTransforms, 0);
        }
        currentTimeValue = time;
        // Read keyframe matrices.
        IList<Keyframe> keyframes = currentClipValue.Keyframes;
        while (currentKeyframe < keyframes.Count)
        {
            Keyframe keyframe = keyframes[currentKeyframe];
            // Stop when we've read up to the current time position.
            if (keyframe.Time > currentTimeValue)
                break;
            // Use this keyframe.
            boneTransforms[keyframe.Bone] = keyframe.Transform;
            currentKeyframe++;
        }
    }
    public void UpdateWorldTransforms(Matrix rootTransform)
    {
        // Root bone.
        worldTransforms[0] = boneTransforms[0] * rootTransform;
        // Child bones.
        for (int bone = 1; bone < worldTransforms.Length; bone++)
        {
            int parentBone = skinningDataValue.SkeletonHierarchy[bone];
            worldTransforms[bone] = boneTransforms[bone] *
                                        worldTransforms[parentBone];
        }
    }
    public void UpdateSkinTransforms()
    {
        for (int bone = 0; bone < skinTransforms.Length; bone++)
```

```
            {
                skinTransforms[bone] = skinningDataValue.InverseBindPose[bone] *
                                       worldTransforms[bone];
            }
        }
        public Matrix[] GetBoneTransforms()
        {
            return boneTransforms;
        }
        public Matrix[] GetWorldTransforms()
        {
            return worldTransforms;
        }
        public Matrix[] GetSkinTransforms()
        {
            return skinTransforms;
        }
        public AnimationClip CurrentClip
        {
            get { return currentClipValue; }
        }
        public TimeSpan CurrentTime
        {
            get { return currentTimeValue; }
        }
}
```

Well, that was a lot of code! Use this class to control the currently playing animation for your model. It has a few pieces of data it needs, starting with the current clip that is played, the current time in the animation, and the current key frame used. It also includes three sets of matrices used to control the animation model and the previously defined `SkinningData` information.

This object is created from a SkinningData object, and the constructor validates the data and creates the arrays of matrices that are used. The next method is `StartClip`, which tells the animation player to start a new set of animation. This method resets the current animation members, and then copies the bind pose transforms to the bone transforms.

As your animation plays, the various matrices need to be updated, so the `Update` method does exactly this. It takes in the amount of time that has elapsed, whether or not the time is relative to the current time, and the root transform of the animation. This in turn calls three helper methods to update the sets of matrices.

The first of these helper methods updates the bone transforms by looking at the list of key frames. First, it checks the time, does a few parameter checks, determines whether the animation has reached the end (and loops back to start if it has), and then begins looking through the key frames. It sets the bone transforms for the key frames required, and then exits the loop. These key frames come from the model during content processing.

Next, it updates the world transforms, which is nothing more than transforming the root bone by the root transform passed in to update, then transforming each child bone by its bone transform and its parent's bone transforms.

For the last of the helper methods, it updates the skin transforms by transforming the inverse bind pose of each bone with the world transform. Finally, there are a few extra methods and properties to expose the private members publicly.

With the shared types out of the way, you can actually create your content processor now to get the animation data into your game.

Go to your `SkinnedModelProcessor` project, remove the code file it created, and add a new one called `SkinnedModelProcessor.cs`. First, add the following code:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline;
using Microsoft.Xna.Framework.Content.Pipeline.Graphics;
using Microsoft.Xna.Framework.Content.Pipeline.Processors;
namespace SkinnedModelProcessor
{
    [ContentProcessor]
    public class SkinnedModelProcessor : ModelProcessor
    {
        public override ModelContent Process(NodeContent input,
             ContentProcessorContext context)
        {
            // Chain to the base ModelProcessor class
            ModelContent model = base.Process(input, context);
            return model;
        }
}
```

Although this isn't exciting, it is the basis for the processor you create next. You created a new content processor, and it is derived from the already existing `ModelProcessor`. In the `Process` method (currently), call the base `Process` method to get all the model data and return it. Although this does nothing more than the `ModelProcessor` (yet), it now has the infrastructure to do so.

To get the animation data through, add the following as the first line of code in your `Process` overload:

```
ValidateMesh(input, context, null);
```

This method validates the mesh, however, it doesn't exist yet, so let's add it to the class:

```
static void ValidateMesh(NodeContent node, ContentProcessorContext context,
                            string parentBoneName)
{
    MeshContent mesh = node as MeshContent;
    if (mesh != null)
    {
        if (parentBoneName != null)
        {
            context.Logger.LogWarning(null, null,
                "Mesh {0} is a child of bone {1}. SkinnedModelProcessor " +
                "does't correctly handle meshes that are children of
➥bones.",
                mesh.Name, parentBoneName);
        }
        if (!MeshHasSkinning(mesh))
        {
            context.Logger.LogWarning(null, null,
                "Mesh {0} has no skinning info, so it has been deleted.",
                mesh.Name);
            mesh.Parent.Children.Remove(mesh);
            return;
        }
    }
    else if (node is BoneContent)
    {
        // If this is a bone, remember that we are now looking inside it.
        parentBoneName = node.Name;
    }
    // Recurse (iterating over a copy of the child collection,
    // because validating children may delete some of them).
    foreach (NodeContent child in new List<NodeContent>(node.Children))
        ValidateMesh(child, context, parentBoneName);
}
static bool MeshHasSkinning(MeshContent mesh)
{
    foreach (GeometryContent geometry in mesh.Geometry)
    {
        if (!geometry.Vertices.Channels.Contains(VertexChannelNames.Weights()))
            return false;
    }

    return true;
}
```

This method uses a helper method defined at the end to determine whether the mesh has skinning information. It does this by looking through each piece of geometry to see whether it has a vertex channel called `Weights`. If it does not, it assumes it has not skinning information. Meanwhile, in the actual method, it determines whether this is a valid mesh.

As it checks the parameters, if it finds problems, it uses the `context.Logger` property to log warnings. This class is part of the content pipeline that runs during build time, so these logged warnings show up in the Visual Studio IDE as warnings during build time.

First, this method checks whether the node is a mesh and if it is, it performs two checks. It determines whether the mesh is a child of a bone and log a warning if so because this processor isn't equipped to handle that scenario. Next, it checks whether the mesh has skinning information and if it does not, it removes itself and logs a warning.

If the current node being checked isn't a mesh, it stores the bone instead because the bone is naturally the parent for all children of this node. After it checks these parameters, this method also recursively iterates over the entire list of children to validate that they're all valid meshes.

In the `Process` override, add the following code after your `ValidateMesh` call to get the animation data:

```
// Find the skeleton.
BoneContent skeleton = MeshHelper.FindSkeleton(input);
if (skeleton == null)
    throw new InvalidContentException("Input skeleton not found.");
// We don't want to have to worry about different parts of the model being
// in different local coordinate systems, so let's just bake everything.
FlattenTransforms(input, skeleton);
// Read the bind pose and skeleton hierarchy data.
IList<BoneContent> bones = MeshHelper.FlattenSkeleton(skeleton);
if (bones.Count > SkinnedEffect.MaxBones)
{
    throw new InvalidContentException(string.Format(
        "Skeleton has {0} bones, but the maximum supported is {1}.",
        bones.Count, SkinnedEffect.MaxBones));
}
List<Matrix> bindPose = new List<Matrix>();
List<Matrix> inverseBindPose = new List<Matrix>();
List<int> skeletonHierarchy = new List<int>();
foreach (BoneContent bone in bones)
{
    bindPose.Add(bone.Transform);
    inverseBindPose.Add(Matrix.Invert(bone.AbsoluteTransform));
    skeletonHierarchy.Add(bones.IndexOf(bone.Parent as BoneContent));
}
// Convert animation data to our runtime format.
Dictionary<string, AnimationClip> animationClips;
animationClips = ProcessAnimations(skeleton.Animations, bones);
```

First, find the skeleton with the built-in helper method `FindSkeleton`. If one isn't found, this model doesn't have the data this processor needs, so an exception is thrown (which shows up as an error in the Visual Studio IDE). After that, use the following `FlattenTransforms` helper method:

```
static void FlattenTransforms(NodeContent node, BoneContent skeleton)
{
    foreach (NodeContent child in node.Children)
    {
        // Don't process the skeleton, because that is special.
        if (child == skeleton)
            continue;
        // Bake the local transform into the actual geometry.
        MeshHelper.TransformScene(child, child.Transform);
        // Having baked it, we can now set the local
        // coordinate system back to identity.
        child.Transform = Matrix.Identity;
        // Recurse.
        FlattenTransforms(child, skeleton);
    }
}
```

This recursively looks through all nodes in this piece of content and transforms the portions of the scene into the same (local) coordinate system. This makes dealing with the various transforms much easier. Next, you get the bind pose of the skeleton and throw an error if it has too many bones for this effect. Then, create the three lists you need to maintain the animation data, and fill them with the appropriate information. Finally, create a set of animation clips to store with the following `ProcessAnimations` helper methods:

```
static Dictionary<string, AnimationClip> ProcessAnimations(
    AnimationContentDictionary animations, IListBoneContent> bones)
{
    // Build up a table mapping bone names to indices.
    Dictionary<string, int> boneMap = new Dictionary<string, int>();
    for (int i = 0; i < bones.Count; i++)
    {
        string boneName = bones[i].Name;
        if (!string.IsNullOrEmpty(boneName))
            boneMap.Add(boneName, i);
    }
    // Convert each animation in turn.
    Dictionary<string, AnimationClip> animationClips;
    animationClips = new Dictionary<string, AnimationClip>();
    foreach (KeyValuePair<string, AnimationContent> animation in animations)
    {
        AnimationClip processed = ProcessAnimation(animation.Value, boneMap);
        animationClips.Add(animation.Key, processed);
    }
    if (animationClips.Count == 0)
    {
        throw new InvalidContentException(
                    "Input file does not contain any animations.");
    }
    return animationClips;
}
```

```
static AnimationClip ProcessAnimation(AnimationContent animation,
                                      Dictionary<string, int> boneMap)
{
    List<Keyframe> keyframes = new List<Keyframe>();
    // For each input animation channel.
    foreach (KeyValuePair<string, AnimationChannel> channel in
        animation.Channels)
    {
        // Look up what bone this channel is controlling.
        int boneIndex;
        if (!boneMap.TryGetValue(channel.Key, out boneIndex))
        {
            throw new InvalidContentException(string.Format(
                "Found animation for bone '{0}', " +
                "which is not part of the skeleton.", channel.Key));
        }
        // Convert the keyframe data.
        foreach (AnimationKeyframe keyframe in channel.Value)
        {
            keyframes.Add(new Keyframe(boneIndex, keyframe.Time,
                                       keyframe.Transform));
        }
    }
    // Sort the merged keyframes by time.
    keyframes.Sort(CompareKeyframeTimes);
    if (keyframes.Count == 0)
        throw new InvalidContentException("Animation has no keyframes.");
    if (animation.Duration <= TimeSpan.Zero)
        throw new InvalidContentException("Animation has a zero duration.");
    return new AnimationClip(animation.Duration, keyframes);
}
static int CompareKeyframeTimes(Keyframe a, Keyframe b)
{
    return a.Time.CompareTo(b.Time);
}
```

The helper methods go through the animations listed on the skeleton and convert them into a list of key frames sorted by time. If there are errors found during this conversion, exceptions are thrown to notify the user during build time; otherwise, the list of animations is returned.

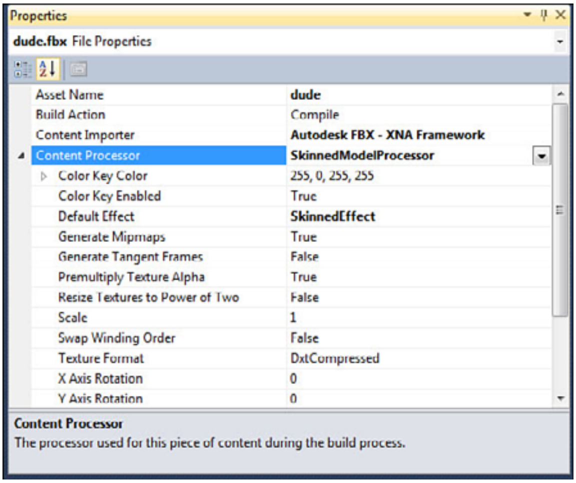
Finally, store this data back into the model before you return it from being processed. So at the end of the `Process` override, before you return the model, add the following code:

```
model.Tag = new SkinningData(animationClips, bindPose,
                              inverseBindPose, skeletonHierarchy);
```

This stores the data you processed during build time into the model object at runtime and in the `Tag` parameter. Build your solution to get the content processor built and ready for use.

Now, you need to add a reference to your content pipeline processor project to your content project. So under the `SkinnedEffectExampleContent` node, right-click the references node, select Add Reference, and choose the projects tab. Then double-click the `SkinnedModelProcessor` project to add the reference.

Now, you're ready for the example, which is simple now. First, add a model to your content project, namely `dude.fbx`. Unlike the models you've added before, change the default processor that this model uses. Under the Content Processor property, choose `SkinnedModelProcessor` from the list, and then open up the tree of properties under this. Under these, change the Default Effect property to SkinnedEffect, which is the effect we're using for this example (see Figure 6.12).



dude.fbx File Properties

Asset Name	**dude**
Build Action	Compile
Content Importer	**Autodesk FBX - XNA Framework**
Content Processor	**SkinnedModelProcessor**
Color Key Color	255, 0, 255, 255
Color Key Enabled	True
Default Effect	**SkinnedEffect**
Generate Mipmaps	True
Generate Tangent Frames	False
Premultiply Texture Alpha	True
Resize Textures to Power of Two	False
Scale	1
Swap Winding Order	False
Texture Format	DxtCompressed
X Axis Rotation	0
Y Axis Rotation	0

Content Processor
The processor used for this piece of content during the build process.

Figure 6.12 Animated model properties

You might be surprised at how little code is required for the rest of this. Because you did a bulk of the work in the content processor (during build time), the runtime components are simple and easy.

> **Note**
>
> The content pipeline is a powerful tool you will want to take advantage of at every opportunity. Anything you can do at build time rather than runtime is a performance saving.

Add the following variables to your game project:

```
Model model;
AnimationPlayer animationPlayer;
Matrix view;
Matrix proj;
```

In your `LoadContent` method, instantiate the following code:

```
model = Content.Load<Model>("dude");
SkinningData skinningData = model.Tag as SkinningData;
if (skinningData == null)
    throw new InvalidOperationException
        ("This model does not contain a SkinningData tag.");
// Create an animation player, and start decoding an animation clip.
animationPlayer = new AnimationPlayer(skinningData);
animationPlayer.StartClip(skinningData.AnimationClips["Take 001"]);
proj = Matrix.CreatePerspectiveFieldOfView(
    MathHelper.PiOver4, GraphicsDevice.Viewport.AspectRatio, 1.0f, 300.0f);
view = Matrix.CreateLookAt(
    new Vector3(2, 45, -110), new Vector3(0,35,0), Vector3.Up);
```

Notice that you use the `Tag` parameter of the model to load the `SkinningData` object that is set as part of the content processing. If this value is null, an error is thrown because it probably means you didn't set the correct content processor. Next, create a new animation player using the data, and start the first clip. In this case, Dude has only one animation named "`Take 001`," and that is what you use. If there are multiple animations, pick the one you want. Finally, set up your camera matrices.

Because animation is happening, you need to have your character's animation update while the game is running—no better place for that than the `Update` method, so add the following to that override:

```
animationPlayer.Update(gameTime.ElapsedGameTime, true, Matrix.Identity);
```

This calls your animation `Update` helper method.You use the elapsed time and have no special root transform, so this is pretty simplistic.The only thing left to do now is to draw, so replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    Matrix[] bones = animationPlayer.GetSkinTransforms();
    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (SkinnedEffect effect in mesh.Effects)
        {
            effect.SetBoneTransforms(bones);
            effect.View = view;
            effect.Projection = proj;
            effect.EnableDefaultLighting();
            effect.SpecularColor = Vector3.Zero;
        }
        mesh.Draw();
    }
}
```

Next, get the list of skin transforms for the current animation, and then loop through each mesh (and each effect on them) to set the bones and the camera information before finally drawing your model. Running the application now is an amazing way to end the chapter; you can see the character walking on the screen (see Figure 6.13).

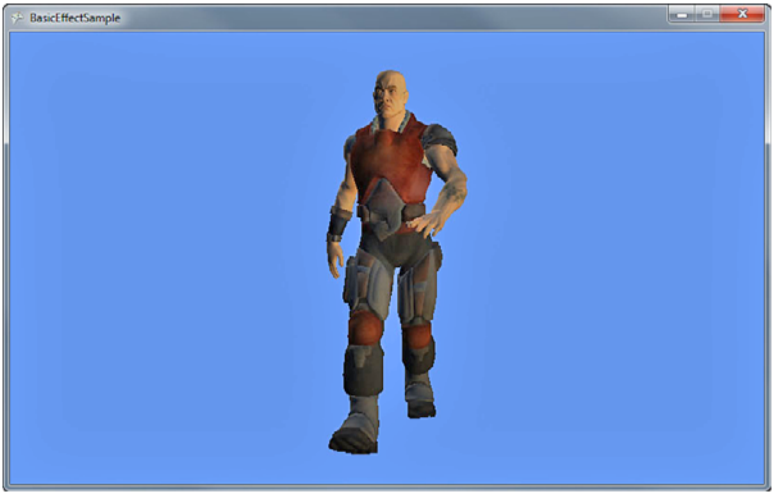


Figure 6.13 An animated character walking

Summary

This chapter contained a lot of information. You learned about built-in effects in the XNA Game Studio framework and the basics of how these effects work. You can do a wide variety of compelling visual techniques with these effects, and they work on all platforms!

In the next chapter, look at blending, textures and the states of the device.

7

States, Blending, and Textures

One of the things you've seen only indirectly is how graphics devices' states and textures interact to form the final output that is rendered to the screen. This chapter delves into this topic more deeply.

Those who are familiar with the previous versions of XNA Game Studio might notice that this is one area where major changes were made. There was a wide variety of reasons, including the fact that it was confusing (and difficult) to ensure the correct values were set, and there were many different options. Although there are still quite a few options available, they're better organized now and much easier to use and set. In this chapter you will learn about:

- Blend States
- Depth States
- Render Targets
- Sampler States

Device States

In previous versions, there were three main objects used to control the graphics device state: namely the `RenderState` object, the `SamplerState` object, and the `StateBlock` object. As the names implied, the first controlled various rendering states, the second sampler states (textures), and the last was a way to control an entire "block" of states.

The `StateBlock` objects looked very useful, but in reality, they were a bit confusing to understand, and the performance of them was abysmal. Even when people used them correctly, the performance hit for using them was enough that they would have been better off not using them at all!

Knowing this, the object type was completely removed for Game Studio 4. Various features that relied on this type (such as the `SaveStateMode` enumeration) were also removed. Don't worry, though, we made using the states themselves so much easier that you won't even miss it being gone.

The `SamplerState` object still exists, although it's behavior and members have changed. The `RenderState` object has changed quite drastically. It has mainly split into three new objects: `BlendState`, `RasterizerState`, and `DepthStencilState`. Again, as the name implies, these objects control the blending and rasterization of the scene, along with the depth and stencil buffer options.

Each of these new state objects has a number of properties that they control, and when they are set on the device, all of the properties are set at once. This enables you to easily set a wide variety of states in a single call, where in previous versions the same operation could take quite a few separate calls.

BlendState

One of the most common states you'll want to change is the `BlendState` object, which controls how each color in your scene blends with another. At a high level, the blend operation is little more than taking two colors, performing some operation on them, and returning a third blended color. The first color is normally called the source color, and the second color is called the destination color. Knowing this, the actual formula for blending is quite simple:

```
output = (sourceColor * sourceBlendFactor) blendFunction
➥(destColor * destBlendFactor)
```

There are several different permutations of this. The following example demonstrates the differences. As they say, pictures are worth a thousand words! Create a new Windows Game project. Add a new item that is a sprite font called Font, and add a piece of content from the accompanying CD, namely alphasprite.png. This is used to easily show the differences in state. Add the following variables to your game class:

```
SpriteFont font;
Texture2D background;
Texture2D alphaImage;
Texture2D red;
```

To load and create these as well, add the following to your `LoadContent` overload:

```
font = Content.Load<SpriteFont>("font");
alphaImage = Content.Load<Texture2D>("alphasprite");
background = new Texture2D(GraphicsDevice, 1, 1);
background.SetData<Color>(new Color[] { Color.CornflowerBlue });
red = new Texture2D(GraphicsDevice, 1, 1);
red.SetData<Color>(new Color[] { Color.Red});
```

This example shows the difference between the various blending modes. To do this, you have one background image (that is the destination color) that is the ubiquitous cornflower blue color and one source image (which includes the source color), which is either a solid red color or the data in the alpha sprite. Replace your `Draw` overload to include the rendering of the background:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(background, GraphicsDevice.Viewport.Bounds, Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

To render your source image easily and compare how it is rendered, add the following method, which helps render the source image with varying blend states:

```
private void DrawBlendedImage(BlendState blend, string text,
    ➥Texture2D image, Point position, Point size)
{
    // Draw the source image on top opaque
    spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.Opaque);
    spriteBatch.Draw(image, new Rectangle(position.X, position.Y,
        size.X, size.Y), Color.White);
    spriteBatch.End();
    Vector2 measure = font.MeasureString(text);
    spriteBatch.Begin();
    spriteBatch.DrawString(font, text, new Vector2(position.X,
        position.Y + size.Y + 2), Color.White);
    spriteBatch.End();
    // Draw the blended version now
    spriteBatch.Begin(SpriteSortMode.Deferred, blend);
    spriteBatch.Draw(image, new Rectangle(position.X,
        position.Y + size.Y + (int)measure.Y + 2,
        size.X, size.Y), Color.White);
    spriteBatch.End();
}
```

There are several sprite batch begins and ends in this code, because you need to draw the images differently so you can tell the difference. After you set the blend mode for a sprite batch, it cannot be changed. Because you draw things with (potentially) different blend modes, you need a separate sprite batch `Begin`/`End` pair for each.

The first call draws the source image using the built-in `BlendState.Opaque` object. The built-in objects are described more in depth in a moment, but for now, this object replaces the destination color with the source, ignoring alpha blending. This enables the source image to be rendered without any blending at all. The source image is drawn at the position passed into this helper method.

Next, draw some text describing which blend operation is shown here. Rendering text requires alpha blending to be enabled, so use the default sprite batch `Begin` call (which uses `SpriteSortMode.Deferred` and `BlendState.AlphaBlend`). Also use the

`MeasureString` call on the font so you know how big it is (so you can start the blended image at the correct spot).

Finally, use your last `Begin`/`End` pair to draw the image with the specified blending state. During this next section of code, you create state objects on the fly, and every frame. Although this certainly works, it is a bad idea. It creates garbage, and unnecessarily makes your game run slower. In a real game, these are cached and reused.

Warning

You should not create new state objects every frame. This example does it for clarity reasons.

Before you create your new blend object and call the helper function, add two more helper functions to easily calculate the point and size parameters you pass in to `DrawBlendedImage`:

```
const int size = 80;
Point GetPoint(int i)
{
    return new Point(10 + ((i % 9) * (size + 6)), 10 + ( (i/9)
➥* (size * 2 + 50) ) );
}
Point GetSize()
{
    return new Point(size, size);
}
```

These simply offset the point by a factor of the size based on the index of the current call. Now add the following code before the `base.Draw` call in your `Draw` overload:

```
int i = 0;
BlendState currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.One;
DrawBlendedImage(currentBlend, "One", red, GetPoint(i++), GetSize());
```

Note

After you set a particular state onto the device, it becomes immutable (you can no longer change it). You need to have separate state objects for each permutation you plan on using.

In the previous formula, the blended color for this image is the following:

```
output = (sourceColor * 1) + (destColor * 1)
```

Essentially, you add the two colors. Can you predict what it would look like without running the example? If you guessed that it would look like a brighter version of the source red, you'd be correct. Let's look at why this is using real numbers in the formula. Assume the source color is RGB (Red, Green, Blue) of (255,0,0) and the destination color is (0,0, 255). You have the following:

```
output = ( (255,0,0) * 1) + ((0,0,255) * 1)
output = (255,0,0) + (0,0,255)
output = (255,0,255)
```

Pure red added with pure blue gives you a purple color. What if your source and destination colors were the same, and they were each (127,127,127) a mid-gray? Using this formula for the colors gives you the following:

```
output = ( (127,127,127) * 1) + ((127,127,127) * 1)
output = (127,127,127) + (127,127,127)
output = (254, 254, 254)
```

This gives you almost a pure white. Adding can be a powerful way to blend. However, what if you wanted to subtract instead? Add this code before your `base.Draw` call in your `Draw` overload:

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Subtract;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.One;
DrawBlendedImage(currentBlend, "SubOne", red, GetPoint(i++), GetSize());
```

Can you predict what this one will do? This is simply the opposite of the addition, so you should see a dark version of the source image. Using the previous examples, if you use both source and destination colors of (127,127,127), it returns pure black (0,0,0). If you use a source color of purple (255,0,255) and a destination of red (255,0,0), it returns pure blue (0,0,255).

Here's an interesting one. Add the following code:

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.ReverseSubtract;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.One;
DrawBlendedImage(currentBlend, "RSubOne", red, GetPoint(i++), GetSize());
```

The `ReverseSubract` value is similar to the Subtract operation except, reversed. At a high level, instead of source-dest, this is dest-source. The formula is the following:

```
output = (destColor * 1) - (sourceColor * 1)
```

Knowing this, can you predict what this output color would be? If you guessed it would appear to be a negative of the source image, then bravo for you again! Let's discuss the last two blend functions. Add the following code:

```
currentBlend = newBlendState();
currentBlend.ColorBlendFunction = BlendFunction.Min;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.One;
DrawBlendedImage(currentBlend, "Min", red, GetPoint(i++), GetSize());
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Max;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.One;
DrawBlendedImage(currentBlend, "Max", red, GetPoint(i++), GetSize());
```

These two functions are basically self-explanatory. It looks at both pixel colors and picks the largest value if you specified `Max`, and the smallest value if you specified `Min`. You see a brighter image for the max function and a darker image for the min function.

Note

The `Max` and `Min` values of the `BlendFunction` enumeration require that both the source and destination use `Blend.One`.

For each of these examples so far, you used Blend.One for both the source and destination. This is easy to understand because it simply leaves the color unchanged; however, there are quite a few different blend operations available. `Blend.Zero` is just as easy to understand, because it multiplies the color by zero, effectively ignoring it (depending on the blend function, of course).

To set the blend operation to simply use the source or destination color, add the following code:

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.SourceColor;
currentBlend.ColorDestinationBlend = Blend.Zero;
DrawBlendedImage(currentBlend, "2xSrc", red, GetPoint(i++), GetSize());
```

Notice that you use the blend operation for the source to be the source color. It doesn't mean that you are simply using the color as is! Remember the formula, plugging in the data gets you the following:

```
output = (sourceColor * sourceColor) + (destColor * 0)
```

Using this essentially doubles the amount of the source in the final color. You can do the same with the following destination:

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.Zero;
currentBlend.ColorDestinationBlend = Blend.DestinationColor;
DrawBlendedImage(currentBlend, "2xDest", red, GetPoint(i++), GetSize());
```

This makes the following formula:

```
output = (sourceColor * 0) + (destColor * destColor)
```

Again, it essentially doubles the destination color. Now, you can also use the inverse of either of these colors by choosing `InverseSourceColor` or `InverseDestinationColor`. Calculate the values by subtracting the current color value from the maximum color value for the channel; for example, if you used single byte

RGB values such as solid green (0,255,0), inversing this color gives you (255 - 0, 255 - 255, 255 - 0) or a purple color of (255,0,255). Using the inverse of a color on itself cancels itself out, as you can see in the following code:

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.InverseSourceColor;
currentBlend.ColorDestinationBlend = Blend.One;
DrawBlendedImage(currentBlend, "InvSrc", red, GetPoint(i++), GetSize());
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.InverseDestinationColor;
DrawBlendedImage(currentBlend, "InvDest", red, GetPoint(i++), GetSize());
```

If you look at the formula, it makes sense (for example, for the first state):

```
output = (sourceColor * inv(sourceColor)) + (destColor * 1)
output = (destColor * 1)
```

However, you can just as easily blend with the opposite colors.

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.DestinationColor;
currentBlend.ColorDestinationBlend = Blend.SourceColor;
DrawBlendedImage(currentBlend, "Swap", red, GetPoint(i++), GetSize());
```

This makes the following formula:

```
output = (sourceColor * destColor) + (destColor * sourceColor)
```

You can even use the inverse of the colors swapped.

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.InverseDestinationColor;
currentBlend.ColorDestinationBlend = Blend.InverseSourceColor;
DrawBlendedImage(currentBlend, "InvSwap", red, GetPoint(i++), GetSize());
```

Let's discuss the last blend operation, `BlendFactor` (and it also has the `InverseBlendFactor` to go along with it). This essentially lets you blend either of your colors with an arbitrary color of your choice, for example:

```
currentBlend = new BlendState();
currentBlend.ColorBlendFunction = BlendFunction.Add;
currentBlend.ColorSourceBlend = Blend.One;
currentBlend.ColorDestinationBlend = Blend.BlendFactor;
currentBlend.BlendFactor = Color.Blue;
DrawBlendedImage(currentBlend, "Factor", red, GetPoint(i++), GetSize());
```

This makes the formula for the output color the following:

```
output = (sourceColor * 1) + (destColor * blue)
```

Given the red color of the source image and the slightly blue background color, this color ends up as a bright purple. A common use of a blend factor is to have it change over time to perform fancy effects.

> **Note**
>
> It is unreasonable to force someone to use a new state object for every blend factor (remember that the state objects are immutable). Therefore, use a `BlendFactor` property on the graphics device to override what is in the state object. Note that the graphics device's value is updated when the state is applied, so make sure you change it only after the state is set.

There are two more properties to discuss briefly. First are the `ColorWriteChannel` properties (there are four total) that specify which color channels can be written during the blend operation. Each channel corresponds to a render target, and by default it writes all the channels.

The `MultisampleMask` property enables you to control which samples are valid for multisampling. By default, all bits are on, so all samples are valid, but you can change this on the fly.

> **Note**
>
> Like `BlendFactor`, `MultisampleMask` is also a property on the device so it can be changed without using the blend state.

Premultiplied Alpha

Another change in this release of Game Studio involves the default mechanism used to perform alpha blending. In previous versions, "normal" interpolated blending was performed for alpha blending, which in the formula you've been using now was defined as:

```
output = (sourceColor * sourceAlpha) + (destColor * inv(sourceAlpha))
```

This was the standard way to declare transparency, but in Game Studio 4.0, the default changed to premultiplied alpha. This mode assumes that the color's RGB values are multiplied by the alpha value before the blend takes place (hence, the term premultiplied), and the formula for this alpha blending is the following:

```
output = sourceColor + (destColor * inv(sourceAlpha)
```

This is similar to the original one, but subtly different enough to matter. Game Studio's defaults changed to use premultiplied alpha because (while it is not perfect) in the majority of cases, it is the blending mode that most developers actually want in the first place. Plus, it is easy to switch back to the old default way of blending.

What does it mean that Game Studio uses premultiplied alpha as the default? Look at the properties of the `AlphaSprite.png` that you added to your content project. Under the texture content processor, notice a property for `PremultipliedAlpha` that is set to true.

This means that during build time when the content pipeline is processing this texture, for each pixel that it processes it premultiplies the alpha in this formula:

```
pixelColor.rgb *= pixelColor.a;
```

Add the following code to your project to see how the alpha computation changes:

```
DrawBlendedImage(BlendState.AlphaBlend, "Alpha", alphaImage, GetPoint(i++),
➥GetSize());
```

If you render this now, the source image has a black background where the alpha channel is when drawn opaque, with a transparent background and the image showing up alpha blended below it. Now, go to the properties of AlphaSprite.Png and change the `PremultipliedAlpha` value to false. Run the application, and notice that the top and bottom images look the same, and rather than a black background, it has a white background with no alpha blending at all.

I'm sure you can guess why. The blend operation expects the alpha value to be premultiplied, and it is not. The background turned white because without the premultiplication multiplying the 0 alpha value with the white RGB value, returning a 0 RGB value of black, it maintains the white color. However, add the following code to the project:

```
DrawBlendedImage(BlendState.NonPremultiplied, "NPAlpha",
  alphaImage, GetPoint(i++), GetSize());
```

Because the image isn't premultiplied anymore, this shows the image normally alpha blended. Lastly, switch the image back to `PremultipliedAlpha` true. Notice that now both premultiplied and non-premultiplied seem to render alpha blended, and appear to look identical. This is because this particular image only has an alpha value of completely on or completely off. There are no partial alpha values, so the two blend operations come up with the same values. This wouldn't be the case otherwise.

Whew, that was a long discussion about blending. There are a number of built-in blend states for the more commonly used operations as well that are static objects off of the `BlendState` itself. These include `Opaque`, `Alphablend`, `Additive`, and `Nonpremultiplied`. After all that, running the app gives you quite a few examples of varying blend modes, much like you see in Figure 7.1.

DepthStencilState

Next up on the list is the depth stencil state. This state has all of the settings and properties required to control the depth and stencil buffers. The previous example is a little dry with just blending the colors, so start a new game project now. Add the `depthmodel.fbx` model to your content project, and add a few variables to the game, too:

```
Model model;
Matrix proj;
Matrix view;
```

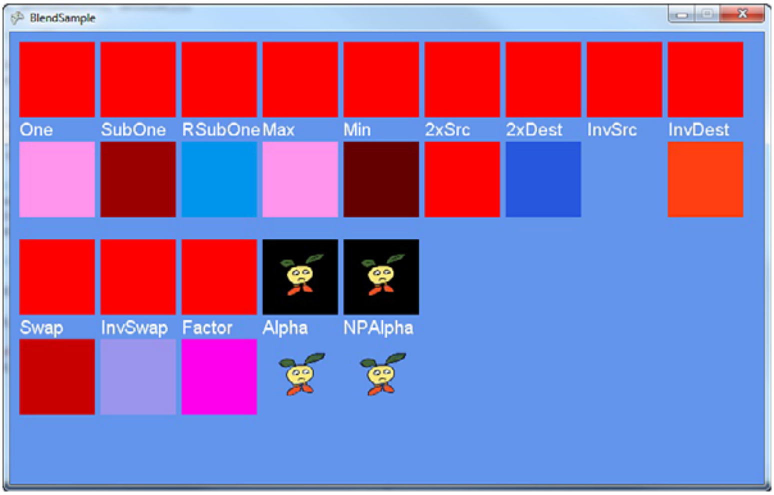

BlendSample
One
SubOne
RSubOne
Max
Min
2xSrc
2xDest
InvSrc
InvDest
Swap
InvSwap
Factor
Alpha
NPAlpha

Figure 7.1 An example of blending states

Load the model and create the matrices, which you can do in your `LoadContent` overload:

```
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
view = Matrix.CreateLookAt(new Vector3(0, 3, 20), Vector3.Zero, Vector3.Up);

model = Content.Load<Model>("depthmodel");
foreach (ModelMesh mm in model.Meshes)
{
   foreach (Effect e in mm.Effects)
    {
        IEffectLights iel = e as IEffectLights;
        if (iel != null)
        {
            iel.EnableDefaultLighting();
        }
    }
}
```

Here, set up a basic camera, load the model, and turn on the lights on the model (because it would look boring and flat otherwise). To draw the model, replace your `Draw` overload with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    float time = (float)gameTime.TotalGameTime.TotalSeconds;
    Matrix rotation = Matrix.CreateRotationZ(time) *
        Matrix.CreateRotationY(time / 4.0f);
    Matrix scale = Matrix.CreateScale(0.5f);
    Matrix transLeft = Matrix.CreateTranslation(-6, 0, 0);
    model.Draw(scale * rotation * transLeft, view, proj);
}
```

This code renders the model to the left of the screen with it slowly rotating around. Notice that the model is a donut-shaped object with four spheres spinning around with it along its outside border. You also should see that at the beginning of the method, you set the `DepthStencilState` property on the device, although you just set it to `Default`. In this particular case, it doesn't matter if you don't make this call. When you draw this model again to the right, it will matter.

What exactly is a depth buffer? Well, it's a buffer to hold depth, of course. If you render a scene with the depth buffer enabled (which is the case by default), for every pixel that is rendered, the depth of that pixel is also stored. You can think of the depth as the distance from the camera. Something near the camera has a depth close to 0.0f, and objects far away have a depth close to 1.0f (the depth runs from 0.0f to 1.0f).

Each time a new pixel is rendered, the system checks whether a depth is written for the position already. If it is, it compares the depth of the current pixel with the one that is already stored, and depending on the function of the depth buffer (which is discussed later), it decides what to do with it. If, for example, the function is `CompareFunction.LessEqual`, it looks at the current pixel's depth. If it is less than or equal to the stored pixel's depth, it writes the new pixel to the buffer; otherwise, it discards it. This enables you to render scenes in 3D, with objects appearing behind (and occluded) by other objects.

To give you an idea of why having the depth buffer is so valuable in a 3D application, add the following code at the end of your `Draw` overload:

```
Matrix transRight = Matrix.CreateTranslation(6, 0, 0);
GraphicsDevice.DepthStencilState = DepthStencilState.None;
model.Draw(scale * rotation * transRight, view, proj);
```

This turns the depth buffer completely off, and then draws the same model again to the right side of the screen, much like you see in Figure 7.2.

Notice that the model on the right looks odd. You can see the small spheres whether they're in front of the torus or not. You can even see some of the inside of the torus through the other portions of itself. With no depth buffer, the last pixel drawn at a particular depth wins.

In the case of this model, it is composed of five different meshes (the torus, and each sphere is its own mesh). The torus is drawn first, and the spheres next, so the spheres always appear on top of the torus, as they're drawn later. Depending on how the torus itself is drawn (and which angle it is at), certain pixels in the center of the torus might be drawn after pixels that would normally block it, so you can also get an odd visual with the center.

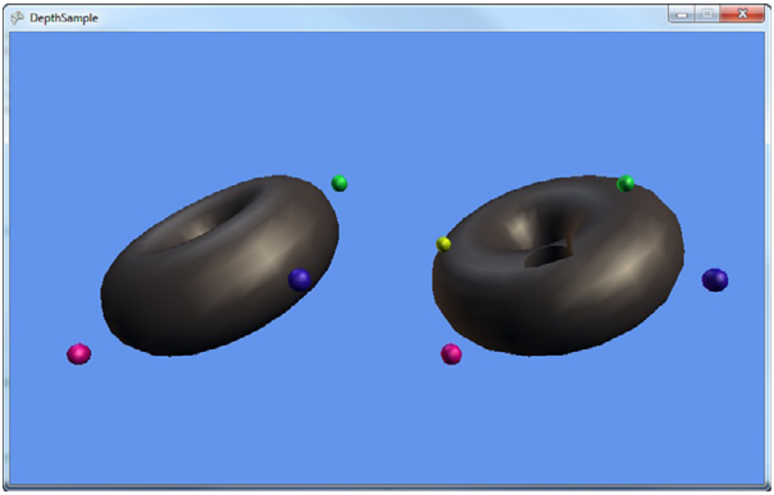


Figure 7.2 Rendering a scene with and without depth

Before moving on to the specific properties of the `DepthStencilState` object, let's take a look at the last static member. You used `Default` (which turns the depth buffer on) and `None` (which turns it off). The last one is `DepthRead`, which reads from the depth buffer, but doesn't write to it. What does that exactly mean?

It means that it does the same operation as the default depth buffer state—it rejects pixels that don't pass the depth test, but when it finds a new pixel that does pass the depth test, it doesn't write the pixel's depth to the buffer. For example, add the following variable to your game:

```
Texture2D grey;
```

Then, create and initialize it in your `LoadContent` method:

```
grey = new Texture2D(GraphicsDevice, 1, 1);
grey.SetData<Color>(new Color[] { Color.Gray });
```

Next, make one change to your current `Draw` overload, namely deleting the line that sets the graphics device to `DepthStencilState.None` before adding this code to the end of the method:

```
GraphicsDevice.Clear(ClearOptions.Target, Color.CornflowerBlue, 1.0f, 0);
model.Draw(scale * rotation, view, proj);
spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend,
    SamplerState.LinearClamp, DepthStencilState.Default,
    RasterizerState.CullCounterClockwise);
spriteBatch.Draw(grey, GraphicsDevice.Viewport.Bounds, null, Color.White,
    0.0f, Vector2.Zero, SpriteEffects.None, 1.0f);
spriteBatch.End();
```

Notice that you're calling a version of `Clear` that you haven't seen before. You're clearing again because you want to erase the two models you just drew, because for this example, they were to write only depth into the depth buffer. However, you cannot use the prototype of `Clear` that you normally use, because it also clears the depth buffer. So instead, tell `Clear` to clear only the `ClearOptions.Target` (the color buffer) to the normal CornflowerBlue, and the rest of the parameters are ignored because they depend on other `ClearOptions` that you didn't specify. You can use `ClearOptions.DepthBuffer` to clear the depth and `ClearOptions.Stencil` to clear the stencil, with the latter two properties as the values to clear each buffer to respectively.

Draw the model again in the center of the screen (this is where you see the effect of `DepthRead` in a few moments).

Next, you use one of the more complex `spriteBatch.Begin` overloads. Aside from the `DepthStencilState.Default`, the rest of the parameters are the defaults for sprite batches though, so you can ignore those for now. Render the texture to cover the entire screen at a layer depth of 1.0f (the last parameter). Specify the `Default` depth state because sprite batch (in its default) turns the depth off. Because you turned it on for this sprite, rendering the texture tests its depth (1.0f) against all the pixels in the depth buffer. Any pixels where the models were drawn have a lesser depth, so the pixels in the texture are discarded and not drawn. As you can see in Figure 7.3, this is exactly what happens because you see a gray screen with the outline of the model's circling there with your extra model floating in between them.

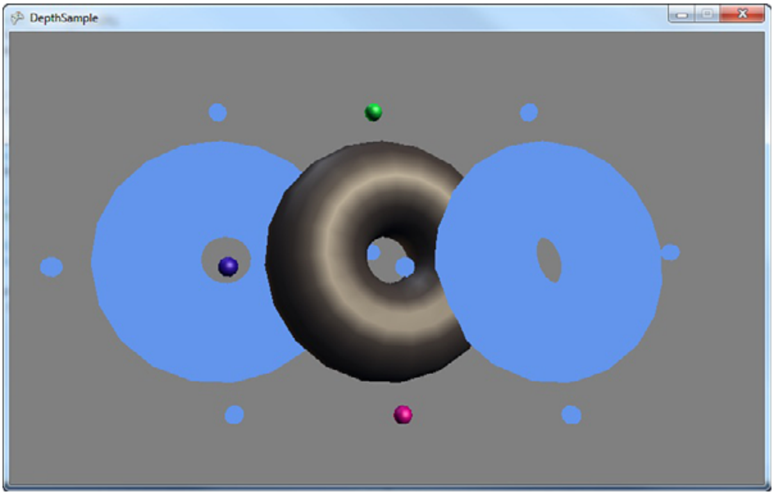


Figure 7.3 Cutouts using depth

Notice that the middle (colored) model looks perfectly normal in the scene. It gets occluded by the cutouts sometimes (when they are behind it), but it shows up in front of the cutouts when it is closer to the camera. This is because it is writing its depth into the depth buffer. Add the following line of code directly after the second `Clear` call:

```
GraphicsDevice.DepthStencilState = DepthStencilState.DepthRead;
```

This is a different output than before. Because it is not writing its depth to the scene, the only places you actually see the rendered model are the portions inside the cutouts where it passed the depth test. So why does this happen?

Let's trace the lifetime of a pixel through this scene. Imagine the pixel is one in which the colored model is drawn through the cutout and intersecting the left model. First, the scene is cleared; the pixel has the default color and a depth value of 1.0f. Next, the left model is drawn, so now the pixel has the left model color in it, and its depth, you can call it 0.1f to have a nice round number. The right model is drawn next, but it doesn't affect the pixel, so now the clear call happens again. This time, the color is reset back to default, but the depth isn't changed, so it's still 0.1f. Now, the colored model is drawn, it has a depth of 0.09f and passes the pixel test, so the color is now updated, but you aren't writing depth, so it is still 0.1f. Finally, the grey texture is drawn, and its depth of 1.0f doesn't pass the depth test, so the colored model's pixel stays there.

If you are looking at a pixel outside of the cutout, though, something else happens. First, the left and right model don't intersect the pixel (otherwise, it would be in a cutout), so this pixel is cleared, has a depth of 1.0f, and then cleared again without modifying the depth. Next, the colored model is drawn; it passes the depth test, updates the pixel color, but hasn't modified the depth, which is still 1.0f. When the sprite comes to render, its value of 1.0f passes the depth test, it overwrites the pixel color with its grey, and you get the scene you see when running this example.

There are only three properties on the `DepthStencilState` object that deal directly with the depth buffer, and they're all basically covered by the static members you've used already. However, because you aren't setting them directly, let's take a look at what they are.

First, you have the `DepthBufferEnable` property, which enables (or disables) the depth buffer completely. The default value for this is true. You should note that everything in this state object is only meaningful if you actually have a depth buffer. If you don't need a depth buffer (for example, a purely 2D game with no layered sprites), you can simply turn the depth buffer off for the entire game by changing your `PreferredDepthStencilFormat` type to `DepthFormat.None` (the default is `DepthFormat.Depth24`) in your constructor:

```
graphics.PreferredDepthStencilFormat = DepthFormat.None;
```

Next is `DepthBufferWriteEnable`, which is by default true. This sets whether the system will take pixels that pass the depth test and write them to the depth buffer. You've seen the behavior when the write is false in the previous example.

The last depth buffer specific property is `DepthBufferFunction`, which has a default of `CompareFunction.LessEqual`. This enables you to dictate how the depth buffer decides whether to allow the pixel or not. By default, the depth buffer is cleared to a value of 1.0f (this is the parameter of the `Clear` function called depth). Because this is the farthest away point in the depth buffer, anything within the viewing frustum passes the depth test at first. However, you can change this behavior with the various values of the `CompareFunction` enumeration.

The rest of the properties deal with the stencil portion of the depth buffer, but first, let's take a bit of time to talk about render targets.

Render Targets

Up until this point, the rendering has been directly onto the screen. You perform a rendering operation, and it appears on the screen. You can go so far as to say that the display was your render target (as in the target of your rendering).

What if you need to render something that you don't need to show on the screen? You would need a different type of render target, and as luck would have it, there is a `RenderTarget2D` class available for you!

A `RenderTarget2D` object is a special kind of texture (it inherits from `Texture2D`) that enables you to use it as a source for rendering, instead of the device. This enables you to use the render target on your device, and then later, use that rendered scene as a texture. To help visualize this concept, create a new Game project and add the depthmodel.fbx from the downloadable examples to the Content project.

Add the following variables to the project so you can draw the model later and use your render target:

```
Model model;
RenderTarget2D rt;
```

Like always, initialize these as well by adding the following code in your `LoadContent` overload:

```
model = Content.Load<Model>("depthmodel");
foreach (ModelMesh mm in model.Meshes)
{
    foreach (Effect e in mm.Effects)
    {
        IEffectLights iel = e as IEffectLights;
        if (iel != null)
        {
            iel.EnableDefaultLighting();
        }
    }
}
rt = new RenderTarget2D(GraphicsDevice, 256, 256, false,
    SurfaceFormat.Color, DepthFormat.Depth16);
```

Load the model and update the lights as you've done many times before. Next is the render target creation. There are three different overloads for creating a render target, with the first as the simplest by taking in the graphics device, the width, and the height. You don't use this one here because a render target created with this overload does not have an associated depth buffer.

The difference between a render target and a texture is render targets can have associated depth buffers. If you think about it, it seems obvious because it can be a target of your rendering, and rendering 3D scenes without a depth buffer can give you ugly scenes.

The overload for creation used here includes the three parameters, and a Boolean parameter to dictate whether the render target includes mipmaps (which are discussed later this chapter), followed by the color format and depth format for the render target. Notice that this example uses a 256×256 render target. Render targets do not have to be the same size as your display or back buffer. This size was chosen because later in the example, it is rendered overlaid on the screen.

Understanding The RenderTargetUsage Options

The last overload includes the parameter to specify `RenderTargetUsage`. This enumeration has three values: `PreserveContents`, `PlatformContents`, and `DiscardContents`. The default value for new render targets is `DiscardContents`, which means whenever a render target is set onto the device its previous contents are destroyed. The framework attempts to help you realize this by clearing the buffer to a solid purple color, a familiar color when working with render targets. If you choose to use `PreserveContents`, then the data associated with the render target is maintained when you switch to and from the render target. Be warned though, that this can have a significant performance impact because, in many cases, it requires storing the data, and then copying it all back into the render target when you use it again. At a high level, the `PlatformContents` chooses between the other two depending on the platform, preserving the data on Windows in most cases, and discarding it on Windows Phone and Xbox 360.

Note

The `RenderTargetUsage` option can be specified for the device as well if you want your device to have preserve semantics.

To show the rendering into the render target, render the model multiple times, and add the following helper method to encapsulate the operation:

```
private void RenderModel(float time, Matrix view, Matrix proj)
{
    Matrix rotation = Matrix.CreateRotationZ(time) *
        Matrix.CreateRotationY(time / 4.0f);
    Matrix scale = Matrix.CreateScale(0.5f);
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    model.Draw(rotation * scale, view, proj);
}
```

This sets up a simple rotating world matrix, turns the depth buffer back to its defaults (you render via the default sprite batch later), and draws the model. This is not fancy, but let's call it. Replace your `Draw` overload with the following:

```
protected override void Draw(GameTime gameTime)
{
    float time = (float)gameTime.TotalGameTime.TotalSeconds;
    Matrix proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
        1.0f, 1.0f, 100.0f);
    Matrix view = Matrix.CreateLookAt(new Vector3(-34, 23, 20),
        Vector3.Zero, Vector3.Up);
    GraphicsDevice.Clear(Color.CadetBlue);
    RenderModel(time, view, proj);
}
```

There's nothing here that's extremely new and exciting, unless of course you consider the different clear color to be new and exciting. Running the example now shows you the model spinning in the center of the screen. Well, the aspect ratio is different too, because you use this (in a moment) on a render target with a 1.0 aspect ratio (width and height are the same). Now, switch to your render target before you call clear using the following line:

```
GraphicsDevice.SetRenderTarget(rt);/
```

> **Note**
>
> You can set up to four render targets at a time for HiDef projects by using the `SetRenderTargets` method instead.

You probably shouldn't run the example now, because you'll just get an exception complaining that you can't call `Present` when a render target is set. To unset the render target when you're done, add the following to the end of your `Draw` method:

```
GraphicsDevice.SetRenderTarget(null);
```

With your rendering now happening onto a separate render target, when you run the example you get a purple screen. As mentioned earlier, this is because your render target (and the device) each have a usage of `DiscardContents`. When a render target is set with this usage, its contents are cleared to a purple color. In this case, when you set the render target back to null (the device's back buffer), its contents are cleared and you see purple. Add the following to the end of your `Draw` method:

```
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
view = Matrix.CreateLookAt(new Vector3(0, 23, 20), Vector3.Zero, Vector3.Up);
GraphicsDevice.Clear(Color.CornflowerBlue);
RenderModel(time, view, proj);
```

Notice that the view matrix has changed so you're looking at the model from a different direction. However, when you run the example, the model spins in the middle of the scene, and there is only one copy of it. It is as if the first version you rendered vanished, which is somewhat true.

Remember earlier when the `RenderTarget2D` object was just a special kind of texture? That means you can use it exactly as you would a texture! So, add the following to the end of the `Draw` method in your example to show both views of the model rendering at the same time:

```
spriteBatch.Begin();
spriteBatch.Draw(rt, Vector2.Zero, Color.White);
spriteBatch.End();
```

There it is! Now you can see that the model you drew originally is now in the upper left corner of the screen as in Figure 7.4, and the second model is still spinning in the center. Also, notice that they're viewed from different angles.

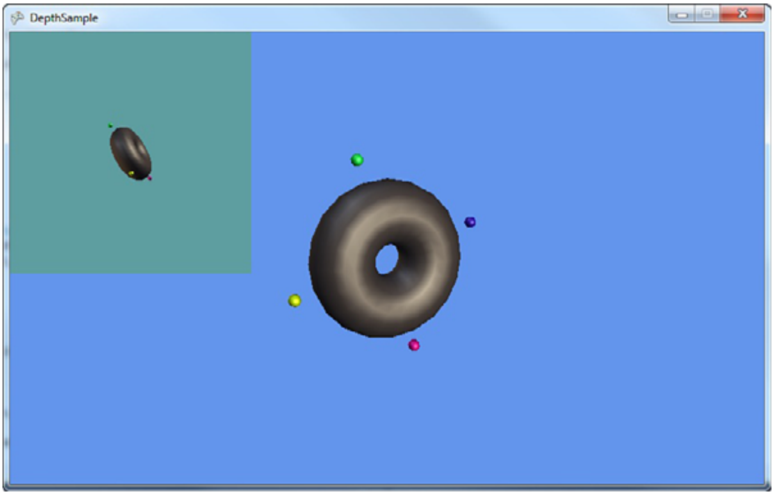


Figure 7.4 Seeing two views from a render target

Note

Just like there is a `RenderTarget2D`, there is also a `RenderTargetCube`, which derives from `TextureCube`.

Faking a Shadow with a Depth Buffer and Render Targets

Why would you want to render something not on the screen? There are quite a few techniques you can use with this type of capability. Let's take a look at one now, by using what you learned so far in this chapter with the depth buffer and the render targets to fake a shadow on a scene. Create a new game project to get started.

The model you are using so far works well for casting shadows, so add `depthmodel.fbx` to your content project. For something to render the shadow onto, add `dualtextureplane.fbx` and `ground.jpg` to your content project. Open the Content

Processor property for `dualtextureplane.fbx`, and change the `Default Effect` property to `DualTextureEffect`, because you will render the shadows as a second texture on the plane. Add the following variables to your game:

```
Model model;
Matrix proj;
Matrix view;
RenderTarget2D lightmap;
Texture2D lightGray;
Texture2D ground;
Model groundModel;
```

The render target object is used to store the shadow data (which manifests itself like the light map in the previous chapter). You also need a small texture to form the cutout of your objects, along with the ground texture. You can initialize these in your `LoadContent` overload:

```
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
view = Matrix.CreateLookAt(new Vector3(0, 23, 20), Vector3.Zero, Vector3.Up);
model = Content.Load<Model>("depthmodel");
foreach (ModelMesh mm in model.Meshes)
{
    foreach (Effect e in mm.Effects)
    {
        IEffectLights iel = e as IEffectLights;
        if (iel != null)
        {
            iel.EnableDefaultLighting();
        }
    }
}
lightGray = new Texture2D(GraphicsDevice, 1, 1);
lightGray.SetData<Color>(new Color[] { Color.LightGray });
ground = Content.Load<Texture2D>("ground");
groundModel = Content.Load<Model>("dualtextureplane");
lightmap = new RenderTarget2D(GraphicsDevice, 512, 512, false,
SurfaceFormat.Color, DepthFormat.Depth16);
```

Most of this should look familiar. Note the size of the render target. The smaller the size, the less memory it takes naturally, but you can also get some visual artifacts for having so few pixels to work with. So instead, a medium-sized render target is created. Lower the size from 512 to something much lower (say 64) after the example is complete, and notice how your shadow appears pixelated. Now, because you render the torus models a few times, add the following helper method to your class:

```
private void RenderScene(float time)
{
    Matrix rotation = Matrix.CreateRotationZ(time) *
        Matrix.CreateRotationY(time / 4.0f);
    Matrix scale = Matrix.CreateScale(0.5f);
    Matrix transLeft = Matrix.CreateTranslation(-6, 0, 0);
    Matrix transRight = Matrix.CreateTranslation(6, 0, 0);
    model.Draw(scale * rotation * transLeft, view, proj);
    model.Draw(scale * rotation * transRight, view, proj);
}
```

This draw your two models side by side—one to the left and one to the right. Now, let's take a few different techniques you've seen over the last couple chapters and combine them into something awesome. Earlier in this chapter, we discussed using the depth buffer and extra clearing to draw a cutout of an object. This is the basis of your shadows, so you can draw these onto your render target. Replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    float time = (float)gameTime.TotalGameTime.TotalSeconds;
    GraphicsDevice.SetRenderTarget(lightmap);
    GraphicsDevice.Clear(Color.CornflowerBlue);
    RenderScene(time);
    GraphicsDevice.Clear(ClearOptions.Target, Color.DarkGray, 1.0f, 0);
    spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend,
        SamplerState.LinearClamp, DepthStencilState.Default,
        RasterizerState.CullCounterClockwise);
    spriteBatch.Draw(lightGray, GraphicsDevice.Viewport.Bounds, null, Color.White,
        0.0f, Vector2.Zero, SpriteEffects.None, 1.0f);
    spriteBatch.End();
    GraphicsDevice.SetRenderTarget(null);
}
```

After setting the render target to a custom one, clear the buffer as normal and call your `RenderScene` helper to draw your two models on the screen. Then, clear the color buffer again (but *not* the depth buffer) to a dark gray color. This is the color of your cutouts, because right after this you draw your simple sprite across the entire render target. If you remember from your `LoadContent` method, your sprite texture is a light gray, which means the render target after this is a light gray solid color with a couple of dark gray cutouts of your model.

Remember that a common usage of `DualTextureEffect` is to render lightmaps onto objects, and your render target now contains essentially just that. Add the following to the end of your `Draw` method to complete rendering of the scene:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
RenderScene(time);
foreach (ModelMesh mesh in groundModel.Meshes)
{
    foreach (DualTextureEffect de in mesh.Effects)
    {
        de.Texture = ground;
        de.Texture2 = lightmap;
    }
}
groundModel.Draw(Matrix.CreateTranslation(0, -20, 0), view, proj);
```

With everything in place, it's simply a matter of rendering your scene again, setting the textures on your `DualTextureEffect` instances on the mesh, and rendering the ground model (that uses those textures) slightly lower in the scene. Your objects now have real-time shadow's as in Figure 7.5.

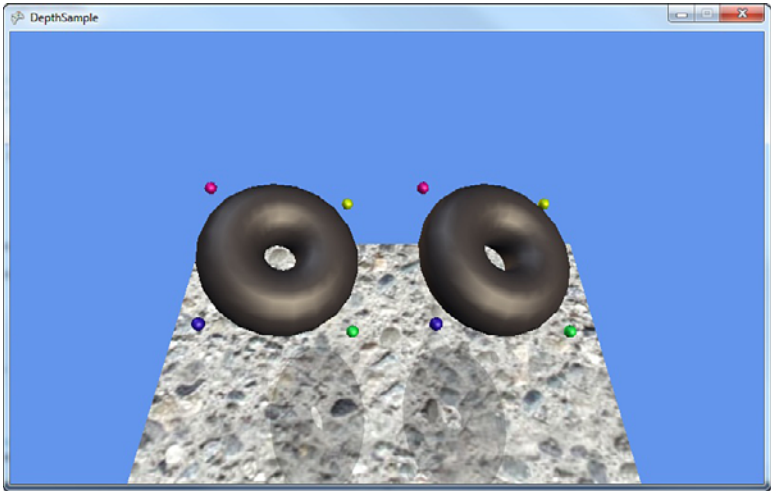


Figure 7.5 Using the depth buffer to form shadows

Back to Device States

Along with the depth buffer, you can also have a portion of that reserved for the stencil buffer. This is similar to the depth buffer in that it is evaluated on a per pixel basis, using a series of tests. Let's look at emulating the cutout you did earlier, but by using the stencil buffer instead.

The Stencil Buffer

Create a new Game project and add the depthmodel.fbx to your Content project. Then, add the following variables to your project:

```
Model model;
Matrix proj;
Matrix view;
Texture2D grey;
DepthStencilState createCutout;
DepthStencilState renderCutout;
```

The last two are the states you use for the stencil buffer—the first to create the cutout, and the second to use the cutout to render. However, by default, your application doesn't even have a stencil buffer; it has only a depth buffer. Go to your game's constructor and add the following line to the end:

```
graphics.PreferredDepthStencilFormat = DepthFormat.Depth24Stencil8;
```

This tells the runtime that you prefer a depth buffer that includes eight bits reserved for the stencil buffer. This is actually the only stencil buffer available in the XNA runtime. With that, instantiate your variables in the `LoadContent` overload:

```
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
view = Matrix.CreateLookAt(new Vector3(0, 3, 20), Vector3.Zero, Vector3.Up);
model = Content.Load<Model>("depthmodel");
foreach (ModelMesh mm in model.Meshes)
{
    foreach (Effect e in mm.Effects)
    {
        IEffectLights iel = e as IEffectLights;
        if (iel != null)
        {
            iel.EnableDefaultLighting();
        }
    }
}
grey = new Texture2D(GraphicsDevice, 1, 1);
grey.SetData<Color>(new Color[] { Color.Gray });
// Initialize some data
createCutout = new DepthStencilState();
renderCutout = new DepthStencilState();
createCutout.StencilEnable = true;
createCutout.StencilFunction = CompareFunction.Always;
createCutout.StencilPass = StencilOperation.Replace;
createCutout.ReferenceStencil = 1;
createCutout.DepthBufferEnable = false;
renderCutout.StencilEnable = true;
renderCutout.StencilFunction = CompareFunction.Equal;
renderCutout.ReferenceStencil = 0;
renderCutout.StencilPass = StencilOperation.Keep;
renderCutout.DepthBufferEnable = false;
```

You should recognize the earlier portion of the snippet, but notice something new after creating the two `DepthStencilState` objects. Before looking at the drawing code (which is quite simple), let's take a look at what these properties are actually doing.

Unlike the depth buffer, the stencil buffer is slightly more complicated than a simple comparison of two values (although, in many cases, it is the same). The basic formula for computing a stencil value is the following:

```
(ReferenceStencil & StencilMask) Function (StencilBufferValue & StencilMask)
```

If your mask value is always all bits, then this is a comparison of the reference value to the buffer value. As you see previously, you use the `ReferenceStencil` property to dictate what the value is. You have the same compare functions that you had with depth buffers, but you have a completely new set of operations you can do if the stencil has passed, found in the `StencilOperation` enumeration. You can use `Replace`, to put the reference value into the buffer, and you can use `Zero`, to put a value of zero in the buffer. You can choose to `Keep` the current buffer value as it is (the default value of the operation). You can choose to `Invert` the value in the buffer, or you can `Increment` or `Decrement` it. The saturate versions of these methods simply clamp the value at the maximum or zero, respectively.

By default, the stencil buffer is turned off, and cleared to a value of zero. So, in order to create the cutout, first turn on the stencil buffer. Then, change the `CompareFunction` to `Always`. This means for every pixel that is drawn, you perform the operation in the `StencilPass` property, which you choose as `Replace`. This replaces the buffer value with the `ReferenceStencil` value, which you've placed as one now.

This means that when you render the models later, using this depth state, the stencil buffer initially is completely zeros, but for every pixel it renders, that pixel's stencil buffer value is updated to one. Next, look at the state you use to render the cutouts.

Again turn on stenciling, but this time set the compare function to `Equal`. Because your `ReferenceStencil` value is zero, any pixel with a stencil value other than zero will fail, and not be drawn. If the stencil value is zero, you keep the value because you specified the `Keep` operation. This means that when you render the second overlay sprite, it does not render the pixels where the model used to be because they have a stencil value of one. Every other pixel has a stencil value of zero.

Now that you have a basic understanding of the stencil buffer and its operation, replace your `Draw` overload with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    GraphicsDevice.DepthStencilState = createCutout;
    float time = (float)gameTime.TotalGameTime.TotalSeconds;
    Matrix rotation = Matrix.CreateRotationZ(time) *
        Matrix.CreateRotationY(time / 4.0f);
    Matrix scale = Matrix.CreateScale(0.5f);
    Matrix transLeft = Matrix.CreateTranslation(-6, 0, 0);
    Matrix transRight = Matrix.CreateTranslation(6, 0, 0);
    model.Draw(scale * rotation * transLeft, view, proj);
    model.Draw(scale * rotation * transRight, view, proj);
    GraphicsDevice.Clear(ClearOptions.Target, Color.CornflowerBlue, 1.0f, 0);
    spriteBatch.Begin(SpriteSortMode.Deferred, null, null, renderCutout, null);
    spriteBatch.Draw(grey, GraphicsDevice.Viewport.Bounds, Color.White);
    spriteBatch.End();
}
```

With that, you now render cutouts of your models using the stencil buffer as in Figure 7.6. It is trivial to update the example you used previously for shadows to mirror this functionality using the stencil buffer.

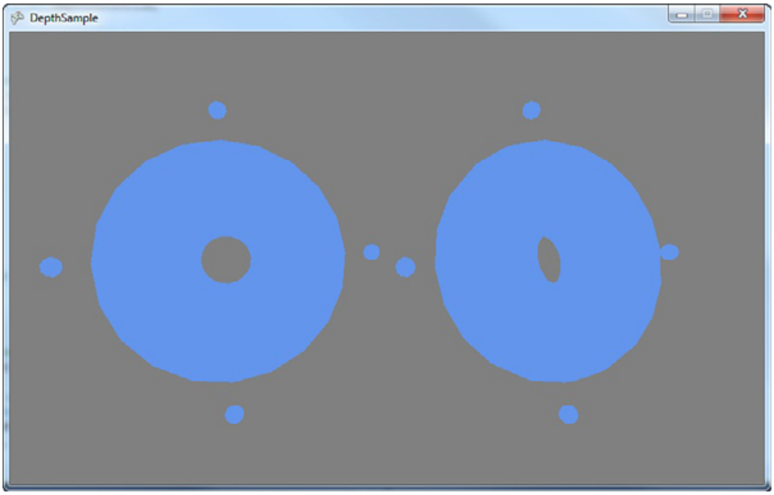


Figure 7.6 Using the stencil buffer to form cutouts

RasterizerState

As you learned in Chapter 4, "Introduction to 3D Graphics," rasterization is where the graphics device converts the triangles into pixels. The `RasterizerState` object enables you to have a measure of control over this process in a variety of ways. To see the kinds of controls you have, create a new Game project and add the depthmodel.fbx file to your Content project. Then, declare your instance variables for the game:

```
Model model;
Matrix proj;
Matrix view;
RasterizerState wireframe;
RasterizerState scissor;
```

The `RasterizerState` class has a few static members that you can use, but you also use two extra states in this example. Create the following objects in your `LoadContent` method:

```
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f);
view = Matrix.CreateLookAt(new Vector3(0, 3, 20), Vector3.Zero, Vector3.Up);
model = Content.Load<Model>("depthmodel");
foreach (ModelMesh mm in model.Meshes)
{
    foreach (Effect e in mm.Effects)
    {
        IEffectLights iel = e as IEffectLights;
        if (iel != null)
        {
            iel.EnableDefaultLighting();
        }
    }
}
wireframe = new RasterizerState();
wireframe.FillMode = FillMode.WireFrame;
scissor = new RasterizerState();
scissor.ScissorTestEnable = true;
```

After the matrices and the model are created and the lighting initialized, create your two needed state objects. The first changes the fill mode to `WireFrame`. The only other option for this property is `Solid`, which is the default. When you're rending an object with `WireFrame` enabled, you see only the triangles rendered, but not the pixels inside of them. For the next option, turn on the `ScissorTestEnable` (the default is false). This is discussed in a moment when you see what it does! Now replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    float time = (float)gameTime.TotalGameTime.TotalSeconds;
    Matrix rotation = Matrix.CreateRotationZ(time) *
        Matrix.CreateRotationY(time / 4.0f);
    Matrix world = Matrix.CreateScale(0.25f) * rotation;
    // Set some rasterizer states
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
    model.Draw(world * Matrix.CreateTranslation(-6, 0, 0), view, proj);
    GraphicsDevice.RasterizerState = wireframe;
    model.Draw(world * Matrix.CreateTranslation(6, 0, 0), view, proj);
    GraphicsDevice.RasterizerState = RasterizerState.CullClockwise;
    model.Draw(world * Matrix.CreateTranslation(0, -6, 0), view, proj);
    GraphicsDevice.RasterizerState = scissor;
    GraphicsDevice.ScissorRectangle = new Rectangle(0, 0,
        GraphicsDevice.Viewport.Width, GraphicsDevice.Viewport.Height / 8);
    model.Draw(world * Matrix.CreateTranslation(0, 6, 0), view, proj);
}
```

Here, you draw the model four different times, each time with a different rasterization setting and in a different location. The first draw call uses the static `RasterizerState.CullCounterClockwise` member, which sets the `CullMode` property of the state to `CullCounterClockwise`. This also happens to be the default, so this first object appears exactly as it normally would if you hadn't set this state. So what exactly does `CullMode` mean, and what does it do?

When the device renders a triangle, it has two sides, but for most 3D models, you can see only one side of a given triangle. So each triangle has a "front" face, the side you expect to see, and a "back" face, the side you probably won't see. The `CullMode` property tells the device to rasterize only pixels for one of those faces. This is determined by the vertices of the triangles "winding order." Triangles have either a winding order of clockwise, or counterclockwise. `CullCounterClockwise` tells the device to cull (remove) faces with a winding order of counterclockwise. The default winding order for XNA applications is that the front-facing triangles are wound clockwise and back-facing triangles are wound-counterclockwise.

Next, draw the model to the right with the wireframe state. As you expect, the model is not drawn solid, but instead with the triangles being drawn, but not the pixels inside of it. This mode is called wireframe.

The bottom model is drawn with the opposite culling mode. While running the application, notice that it looks odd. This is because it's rendering only the "wrong side" of each triangle, so you see what the model looks like almost inside out. You can also use the built-in static member `RasterizerState.CullNone` to render both sides of each triangle.

The last model (the one drawn at the top of the screen) simply turns on the scissor test. The scissor test tells the device to not render any pixels that are outside the `ScissorRectangle` property of the device (which by default is the same size as the currently applied render target or back buffer). In this example, you set the rectangle to be the entire width of the screen, but only the upper eighth portion, which causes the model to be cut in half. Why are the other three models still showing despite the fact you told the device to render only in the upper eighth of the screen? The other three models were rendered with a state that had the scissor test turned off.

When you run the example, you see each of the four rasterizer states, as shown in Figure 7.7.

SamplerStates

The last of the device states we discuss is the sampler state. This state controls how the textures that are rendered by the system are handled. Why is it called a sampler state rather than a texture state? These settings are used when the texture is actually sampled to get a particular color, rather than the state of the texture itself.

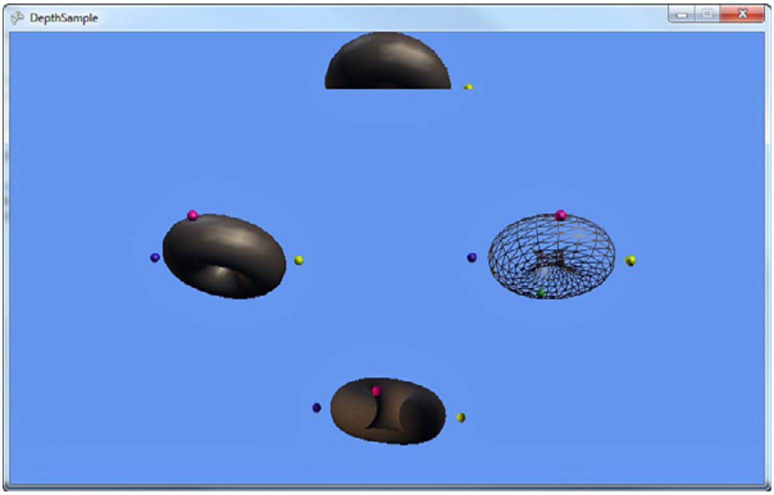


Figure 7.7 Various rasterizer settings

The majority of pre-created static members of the object have to do with texture addressing. Normally, texture coordinates go from the range of 0 to 1, but what if yours go beyond that? The texture addressing mode determines how the texture is sampled from outside this range, so let's take a look at a quick example. Begin by creating a new Game project and adding a texture to the Content project (for example, cat.jpg). Declare a variable to hold your texture and one of your states as well:

```
Texture2D texture;
SamplerState linearMirror;
```

Initialize the following variables in your `LoadContent` method as normal:

```
texture = Content.Load<Texture2D>("cat");
linearMirror = new SamplerState();
linearMirror.AddressU = TextureAddressMode.Mirror;
linearMirror.AddressV = TextureAddressMode.Mirror;
linearMirror.AddressW = TextureAddressMode.Mirror;
linearMirror.Filter = TextureFilter.Linear;
```

Before we go on, let's take a look at the state you just created. As you can tell, you can set the address mode on any of the components of the texture coordinates separately (U,V, or W), although, in most cases, you simply set them all to the same thing as you did here. We discuss what each of the three valid address modes are in a moment after you see the other two. Replace the `Draw` overload with the following and run the application:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    int seconds = (int)gameTime.TotalGameTime.TotalSeconds % 3;
    SamplerState state = (seconds == 0) ? SamplerState.LinearClamp :
        ((seconds == 1) ? SamplerState.LinearWrap : linearMirror);
    spriteBatch.Begin(SpriteSortMode.Deferred, null, state, null, null);
    spriteBatch.Draw(texture, GraphicsDevice.Viewport.Bounds,
        new Rectangle(-GraphicsDevice.Viewport.Width, -
GraphicsDevice.Viewport.Height,
            GraphicsDevice.Viewport.Width * 3, GraphicsDevice.Viewport.Height * 3),
            Color.White);
    spriteBatch.End();
}
```

At the beginning of this method, you first pick which sampler state you will use, using a quick formula to change it every second between one of three different states. The first is `LinearClamp`, which is just like the one you created previously, except `Mirror` replaced with `Clamp` as the addressing mode. Next is `LinearWrap`, which again, uses the `Wrap` addressing mode. The final one is the mirror mode you created earlier. What do each of these do? You can probably tell partially just by running the example, which rotates through them as in Figure 7.8.

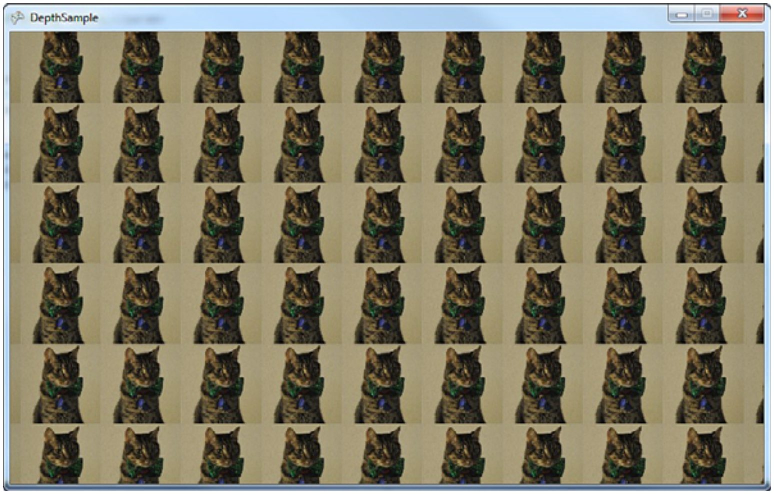


Figure 7.8 The Wrap texture addressing mode

The first mode is `Clamp`. This "clamps" your texture coordinates between 0 and 1. It does this by artificially stretching the border pixels for the texture infinitely. This is why when you see the cat rendering from the beginning, it looks like it is smeared into position where you can bands to the left and right, as well from the top and bottom of the texture.

`Wrap`, which you see during the running example (and can somewhat tell from the name), simply wraps the texture indefinitely in all directions. A UV coordinate of 1.5,1.5 is read the same as a UV coordinate of 0.5,0.5.

The last mode is the `Mirror` mode, which is similar to `Wrap` in that it repeats the image indefinitely, but each new image required is flipped from the previous one. Notice that the flipping happens on both the horizontal and vertical axis as well.

You might wonder how this image was rendered with a set of texture coordinates outside of the 0,1 range. After all, you expect that using sprite batch renders only the texture you passed in, and for the majority of cases, this is very true. However, look at the third parameter to the sprite batch `Draw` method here. You use a source rectangle that goes beyond the bounds of the texture. The sprite batch detects this and maps the UV coordinates across the range of the source rectangle, where the ranges from 0–1 are the coordinates within the actual source of the texture, and everything else is not.

The last parameter is the Filter of the sampler, which dictates how the sample color is determined if it isn't exactly a single pixel (for example, it is between two pixels). The XNA Game Studio runtime supports three different filters: Linear, Point, and Anisotropic. Notice a few other permutations in the list of possible values with various combinations of "min" (standing for minify, the act of shrinking a texture), "mag" (standing for magnify, when a texture is enlarged), and "mip" (standing for the mip levels, when you have a texture with a set of mip maps). For example, the `MinLinearMagPointMipLinear` value means it uses linear filtering during minify, point filter during magnify, and linear filter during mipmapping.

What Is Mipmapping?

At a high level, mipmaps are a collection of images representing a single texture at various depths. Each image is half the size of the one before it. For example, if you have a starting image that is 256×256 pixels, it has a series of eight different additional images for different depths (or mip levels), of sizes 128×128, 64×64, 32×32, 16×16, 8×8, 4×4, 2×2, and 1×1.

If your texture has mipmaps, the runtime picks the appropriate sized image based on the distance from the camera. Items extremely close to the camera use the higher resolution images, and items far from the camera use the low resolution images. Normally, mipmaps are generated by filtering the next higher resolution down. However, there's nothing stopping you from having your artist give you the entire mipmap chain and using it directly.

Although you can get a boost in image quality by using mipmaps, one of the major disadvantages is memory consumption. Because you store multiple different copies of the image at different resolutions, you use at least double the memory.

Other Texture Types

Although you've been using mostly `Texture2D` and similar objects, there are actually two other types of textures available in the system as well. These are the `TextureCube` and `Texture3D` classes.

A cube texture (like the name implies) is a texture that includes six sides, such as in a cube (refer to Chapter 6, "Built-In Shader Effects"). A common usage of this texture type for the environment map effect is to render portions of the scene six times (one for each face of the cube) into a cube render target and then use it as your environment map.

> **Note**
>
> `Texture3D` are available only in the HiDef profile.

Summary

Unlike previous versions of Game Studio, you are encouraged to update your state whenever you draw something new. Setting state is cheap and easy, and now you have the basics for how to use the various permutations available to you. If you ever want to get the device back to its default state, you can do it in four easy lines:

```
GraphicsDevice.BlendState = BlendState.Opaque;
GraphicsDevice.DepthStencilState = DepthStencilState.Default;
GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;
```

In the next chapter, you delve into the world of custom effects, where you write effects to do amazing things.

8

Introduction to Custom Effects

If you want the ultimate control over the graphics in your game you will want to write your own custom shader effects. Writing your own effects allows you the developer to control how the graphics hardware renders the geometry in your game.

In this chapter you will learn:

- The different parts that make up a custom effect file
- How to create your first custom effect file
- Vertex coloring and texturing
- Numerious lighting models for directional and point lights
- Setting effect states using an effect

Although XNA Game Studio 4.0 provides five built-in effects to use in your game, you are not limited to just these five for your Windows and Xbox 360 games. Underneath the five built-in effects are compiled effects graphics programs that implement the features that are exposed by the different types.

What Is a Custom Effect?

Custom effects enable you to write an effect file that contains the vertex and pixel shaders that determine how the geometry you draw is transformed and shaded before being outputted to the screen. This gives you tremendous control over how geometry is rendered in your game. With this control also comes some additional difficulty in programming and debugging your graphics code.

Although custom effects work in both the Reach and HiDef profiles, custom effects are not supported on Windows Phone 7.

High Level Shading Language

Effect files in XNA Game Studio are written in the High Level Shading Language or HLSL for short. HLSL was created by Microsoft for use by applications written using Direct3D. The underlying XNA graphics stack is built on Direct3D and thus uses HLSL as the effect shading language.

The syntax of HLSL is similar to C but with additional language support specifically designed for graphics operations.

HLSL supports many of the common keywords found in C such as `bool`, `int`, `if`, and `else` but also have specific keywords such as `sampler`, `technique`, and `pixelshader`.

HLSL also supports built-in intrinsic functions that are used in many graphics algorithms. For example, the `dot(a, b)` intrinsic returns the dot product scalar value between two vectors `a` and `b`.

HLSL has multiple versions, which can differ from the number of compiled instructions that can be used, the number of textures that can be accessed, to the actual keywords and intrinsic functions that are available. These versions are called shader models and can differ from the vertex to the pixel shaders. If your game is targeting the Reach profile, you can use the 2.0 shader model version. The HiDef profile supports shader model 3.0+.

> **Note**
>
> The Xbox 360 supports a special shader model version of 3.0, which includes instructions that are not available on other platforms such as `vfetch`. The list of available instructions can be found in the XNA Game Studio documentation.

This chapter is focused on getting you started on writing your first few effects and does not cover HLSL in depth. There are many books dedicated just to the topic of writing effects and is beyond what can be covered in just one chapter. You receive a good foundation to start experimenting with your own effects and will have the basic understanding so you can read about other types of effects.

Creating Your First Custom Effect

Let's start with a new effect file. Effect files generally end with the .fx extension. Create a new XNA Game Studio Windows project. Then right-click the content project and select Add -> New Item (see Figure 8.1).

The Add New Item dialog displays a few of the different content types supported by XNA Game Studio. Select the Effect File listing and name it `CustomEffect.fx`. Finally, click the Add button (see Figure 8.2).

The `CustomEffect.fx` file is not added to your content project. Effect files are built using the content pipeline using the Effect importer and processor. Because effect files contain code that is run on the graphics processor, the code needs to be compiled into a binary format suitable to run on the graphics hardware. This is similar to how a

C application must be compiled to run on your Windows PC. The content pipeline compiles your effect file and reports build errors. Just like your game code, the effect file can contain syntax errors that are reported in the build output window in Visual Studio.

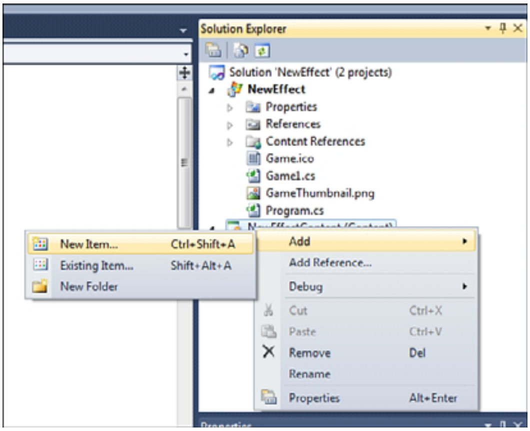

Solution Explorer
Solution 'NewEffect' (2 projects)
NewEffect
Properties
References
Content References
Game.ico
Game1.cs
GameThumbnail.png
Program.cs
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Add
Add Reference...
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Properties Alt+Enter

Figure 8.1 Add new content item menu

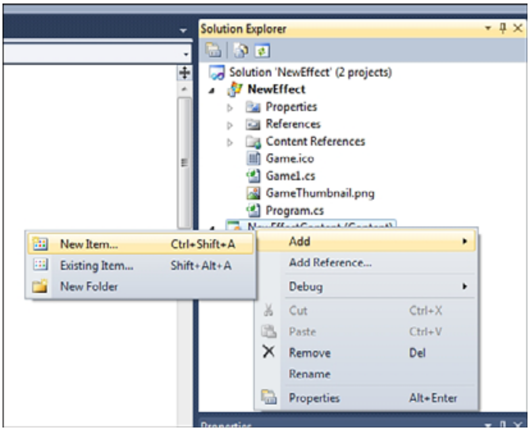

Solution Explorer
Solution 'NewEffect' (2 projects)
NewEffect
Properties
References
Content References
Game.ico
Game1.cs
GameThumbnail.png
Program.cs
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Add
Add Reference...
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Properties Alt+Enter

Figure 8.2 Add New Item dialog

Parts of an Effect File

Double-click the newly created CustomEffect.fx file to view the contents of the file in the Visual Studio editor. Notice that the newly created file is not empty. The file is filled with the default effect file template to help you get started writing your effect.

Global Variables

The first part of the effect template declares three global variables that can be used from within the shaders you write.

```
float4x4 World;
float4x4 View;
float4x4 Projection;

// TODO: add effect parameters here.
```

The `World`, `View`, and `Projection` matrices are used to transform the geometry from local space into screen space as described in Chapter 4, "Introduction to 3D Graphics." Notice that the variable type is not matrix, but is instead `float4x4` (pronounced float four by four). This is essentially a four-by-four array of floats, which can be used to represent a 4×4 matrix. The TODO comment lets you know that this is the location where you can add new global parameters to the effect file.

Vertex Structures

The next section of the template defies two structures.

```
struct VertexShaderInput
{
    float4 Position : POSITION0;

    // TODO: add input channels such as texture
    // coordinates and vertex colors here.
};

struct VertexShaderOutput
{
    float4 Position : POSITION0;

    // TODO: add vertex shader outputs such as colors and texture
    // coordinates here. These values will automatically be interpolated
    // over the triangle, and provided as input to your pixel shader.
};
```

These structures are used to define what the input and output types are for the vertex shader. Refer to Chapter 4 where we discuss the graphics pipeline the vertex shader is used to transform the geometry from local space into screen space.

Input Vertex Structure

The first structure `VertexShaderInput` defines how the geometry will be passed into the shader. In this case, only the position of the geometry is passed into the shader with the name `Position` and has a type of `float4`. Notice that to the right of the variable is `POSITION0`. This is an input semantic that defines what each field represents in the vertex.

It is used to define which portions of the input vertex stream correspond to the fields of the vertex structure in the shader.

When geometry is drawn in a game, a `VertexDeclaration` must be specified in order for the graphics hardware to determine how each field in a vertex element is used. The graphics hardware can then pass each of the vertices into the vertex shader.

The type used by the vertex in your game and the type used in the shader don't have to match. The graphics card tries to map the vertex data the best it can given the hints from the semantics. For example, your `VertexBuffer` might contain the positions as `Vector3` values, which contains three floats, but your input vertex structure might be expecting a `float4`. The first three components will be copied into the input vertex structure and the final w component will be left blank.

Output Vertex Structure

The output vertex structure is used to define what values are passed out of the vertex shader and into the pixel shader. Although the vertex shader is run for each geometry vertex that is drawn, the pixel shader is run for each pixel that makes up the triangle that is drawn.

The output vertex structure is often used as the input structure for the pixel shader, but this is not required. What is required is for the vertex shader to output a position. The position is required because it is used to determine how the triangle should be displayed on the screen. If portions or all of the triangle corners are not going to be displayed on the screen, then the pixel shader is not drawn for these pixels.

You might already notice something of importance. Although the vertex shader is run exactly once per vertex, the pixel shader can be run from 0 to the total number of pixels on the screen for each triangle. So there is no one-to-one relationship between the vertex and pixel shaders. How does the output from the vertex shader correspond to the input of the pixel shader when it can be run many more times than the vertex shader?

The answer is that the values from the vertex shader do not directly pass into the pixel shader. The values first pass through an interpolater. The interpolater does exactly what its name describes, which is to interpolate values across the face and edges of the triangle.

After the vertices of the triangle are passed through the vertex shader, the pixels that make up the triangle are determined and the pixel shader is run for each of the pixels. The interpolator is used to determine the values between each vertex. For example, if one vertex has and X position of 0.1 and another has an X position of 0.5, then the pixel that is halfway between the two on the edge of the triangle has an X value of 0.3.

Along with the position values, other values can be interpolated across the triangle such as color if the vertices or texture coordinates.

Vertex Shader

The next section of the effect template defines the vertex shader.

```
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;
    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);

    // TODO: add your vertex shader code here.

    return output;
}
```

This is a very simple vertex shader. A new instance of the `VertexShaderOutput` structure is defined and used to return the final position at the end of the shader. The vertex shader then takes the input position and multiples it by the `World` matrix first to move the position into world space. It then multiplies the position by the `View` to move the position into view space. Finally, it multiplies the position by the `Projection` to move the it into projection or clip space.

Pixel Shader

The output position from the vertex shader determines all of the pixels that are displayed on the screen. The pixel shader is then run for each of the pixels.

The following simple pixel shader is defined by the new effect template.

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
    // TODO: add your pixel shader code here.

    return float4(1, 0, 0, 1);
}
```

Notice that the input to the pixel shader function takes an argument of the type `VertexShaderOutput`. Also notice that the output of the pixel shader is a `float4` that represents the color to write to the color buffer. The `COLOR0` semantic is used to tell the graphics hardware the pixel shader plans to output the color using the `float4` return value of the shader.

This simple shader outputs a solid color of red. A `float4` color is storage in RGBA format. This means that the `X` component of the `float4` is the red channel. The green, blue, and alpha values are stored in the `Y`, `Z`, and `W` components respectively.

Techniques and Passes

The last section of the effect template defines the techniques and the passes they contain.

```
technique Technique1
{
    pass Pass1
    {
        // TODO: set renderstates here.
        VertexShader = compile vs_2_0 VertexShaderFunction();
        PixelShader = compile ps_2_0 PixelShaderFunction();
    }
}
```

A technique is a collection of passes that are designed to run for each triangle. An effect file must contain at least one technique, but it can contain many techniques. Multiple techniques can be used when you need to alter which vertex and pixel shaders to use for particular geometry. For example, multiple techniques are used in the five built-in effects to handle the different types of input vertices that are drawn using the effects.

An effect pass defines which vertex and pixel shader should be used by the graphics hardware. The shader version along with the `compile` keyword determine how the vertex and pixel shaders should be compiled. The default template uses shader model 2 as the default, so `vs_2_0` is used for the vertex shader and `ps_2_0` is used for the pixel shader.

Each effect pass can also contain a set of render state to set before drawing geometry. Like the state objects discussed in Chapter 7, "States, Blending, and Textures," these settings enable another method to set the graphics device settings to use while rendering geometry.

This brings you to the end of the new effect file template. It is not too complex, but this program runs many times very quickly on the graphics hardware. The complexity of your effects has a direct influence on the performance of drawing geometry using your effect. Some operations are more expensive than others, but a general tip is to keep the number of operations to a minimum.

Drawing with a Custom Effect

So you have your first custom effect file but nothing to show for it on the screen. Let's get started by drawing a quad with the new effect file.

Define the following two member variables to hold the `Effect` and the array of vertices make up the two triangles in the quad.

```
Effect customEffect;
VertexPositionColor[] userPrimitives;
```

The `Effect` type is used with custom effects and is the base type of the five built-in effects. There are a number of built-in vertex types. The `VertexPositionColor` contains both the vertex position and the color for the vertex. All built-in vertex types implement the `IVertexType` interface that requires the vertex type to specify the `VertexDeclaration`. As we discussed before, the `VertexDeclaration` is used to tell the graphics hardware how the fields of the vertex should be used, such as position and color.

Next, in the games `LoadContent` method, load the `Effect` and define each of the vertices.

```
// Load custom effect and set parameters
customEffect = Content.Load<Effect>("CustomEffect");
customEffect.Parameters["World"].SetValue(Matrix.Identity);
customEffect.Parameters["View"].SetValue(Matrix.CreateLookAt(new Vector3(0, 0, 3),
➥new
Vector3(0, 0, 0), new Vector3(0, 1, 0)));
customEffect.Parameters["Projection"].SetValue(Matrix.CreatePerspectiveFieldOfView
➥(MathHel
per.PiOver4, GraphicsDevice.Viewport.AspectRatio, 0.1f, 100.0f));

// Create the verticies for our triangle
userPrimitives = new VertexPositionColor[4];
userPrimitives[0] = new VertexPositionColor();
userPrimitives[0].Position = new Vector3(-1, 1, 0);
userPrimitives[0].Color = Color.Red;
userPrimitives[1] = new VertexPositionColor();
userPrimitives[1].Position = new Vector3(1, 1, 0);
userPrimitives[1].Color = Color.Green;
userPrimitives[2] = new VertexPositionColor();
userPrimitives[2].Position = new Vector3(-1, -1, 0);
userPrimitives[2].Color = Color.Blue;
userPrimitives[3] = new VertexPositionColor();
userPrimitives[3].Position = new Vector3(1, -1, 0);
userPrimitives[3].Color = Color.Yellow;
```

Notice that the `Effect` is loaded through the content pipeline. After loading the effect, set three properties for the `World`, `View`, and `Project` matrices. Although the built-in effects expose these as properties in the API surface, your `Effect` can't because there is no guarantee that the `Effect` will use those global variables. Instead, use the `Parameters` collection and specify the parameter by string name.

Finally, in the games `Draw` method, tell the graphics card that you are using this specific effect and to draw the two triangles.

```
// Start using the BasicEffect
customEffect.CurrentTechnique.Passes[0].Apply();
// Draw the primitives
GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(PrimitiveType.TriangleStrip,
                                                       userPrimitives, 0, 2);
```

Before using an `Effect`, call `Apply` on the specific `EffectPass` you plan to use. In this case, you know there is only one `EffectTechnique`, so use the `CurrentTechnique` and there is only one `EffectPass`, so use index 0.

Now you are ready to draw. Press F5 and see the glory that is your first effect. Figure 8.3 shows an example of what you should see.

Well that's not too exciting, but it draws a pure red quad to the screen. Notice that the colors that were defined for each vertex are not used because the template pixel shader just outputs the single red color and does not use the vertex color. Don't worry about it—it gets better looking from here.

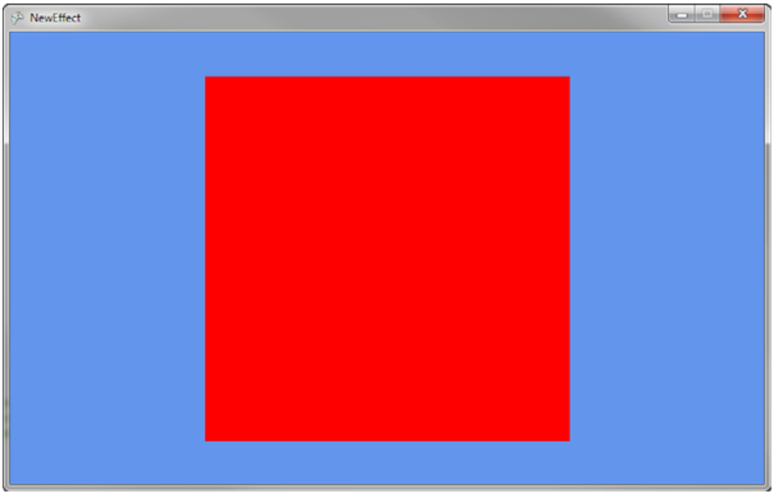


Figure 8.3 Drawing a quad with your first custom effect

Vertex Color

Now it's time to add a little color to the quad. The last example outputted a single red color for each pixel. Let's take the color for each vertex and pass it through the pixel shader so it can output the color.

The vertices that make up the quad have colors defined at each vertex, so the input data is already available—you are not taking advantage of the colors in the effect file.

To add vertex color support to the effect, update the `VertexShaderInput` to include a new field for the color.

```
struct VertexShaderInput
{
    float4 Position : POSITION0;
    float4 Color    : COLOR0;
};
```

The `VertexShaderOutput` also needs to be updated with the new Color field.

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float4 Color    : COLOR0;
};
```

In the vertex shader, copy the input color into the output structure.

```
output.Color = input.Color;
```

The color interpolates across the triangles and blends as the color changes from one vertex to another.

Finally, the color needs to return from the pixel shader. Remove the existing return statement and replace it with the following line:

```
return input.Color;
```

Running the game now displays the quad with four unique colors for each vertex. The colors are interpolated across the face of the two triangles and should look like Figure 8.4.

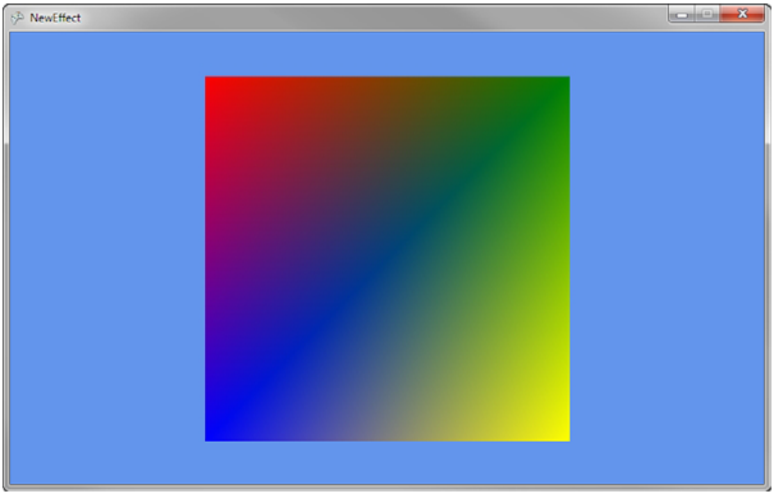


Figure 8.4 Quad rendered with vertex coloring

Texturing

To give the triangles a little more detail, you are going to remove the color from the triangles and replace them with a texture to map across each of the two triangles.

Textures are mapped to triangles using texture coordinates. Each vertex gets a specific texture coordinate to tell the graphics hardware that at this vertex, the following coordinate should be used to sample from the texture. No matter the pixel size of a texture, the texture coordinates are from 0 to 1 in both width direction called `U` and the height direction called `V`. Position 0,0 or the origin of the texture is the top left of the texture, and position 1,1 is the bottom right of the texture.

Note

Volume textures have an additional axis for depth, which uses the `W` component.

Next, you update the existing sample to use a texture instead of vertex colors. Add any texture to your content project. For example, add Fractal.png.

Note

Constraints are placed on textures depending on the `GraphicsProfile` that you have specified for your game. The `HiDef` profile can support texture sizes up to 4096, and textures in the `Reach` profile must be 2048 or smaller in size. In addition, `Reach` textures don't support the wrapping, mipmaps, or DXT compression if the texture is not a power of two.

In your game, change the vertex type used to store the vertices and declare a member variable to store the texture you plan to render on the quad.

```
VertexPositionTexture[] userPrimitives;
Texture2D colorTexture;
```

You have removed the `VertexPositionColor` type and replaced it with the `VertexPositionTexture`. This vertex type includes a field to use for the texture coordinate.

In the game's `LoadContent` method, load the texture and set the texture onto the device.

```
// Load the texture we want to display
colorTexture = Content.Load<Texture2D>("Fractal");
GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;
GraphicsDevice.Textures[0] = colorTexture;
```

You have seen the first line previously. Load the texture like normal through the content pipeline and store it in a `Texture2D` variable instance.

Then, set a `SamplerState` for the first texture sampler on the `GraphicsDevice`. As discussed in Chapter 7, the `SamplerState` tells the graphics hardware how to sample the texture as it textures any triangles. The final line of code sets the texture the `GraphicsDevice` should sample from when drawing.

Because you are now using a new vertex type, update how you declare the array of vertices for the quad you are drawing. Remove the existing list of vertices in the `LoadContent` method and add the following lines of code:

```
// Create the verticies for our triangle
userPrimitives = new VertexPositionTexture[4];
userPrimitives[0] = new VertexPositionTexture();
userPrimitives[0].Position = new Vector3(-1, 1, 0);
userPrimitives[0].TextureCoordinate = new Vector2(0, 0);
userPrimitives[1] = new VertexPositionTexture();
userPrimitives[1].Position = new Vector3(1, 1, 0);
userPrimitives[1].TextureCoordinate = new Vector2(1, 0);
userPrimitives[2] = new VertexPositionTexture();
userPrimitives[2].Position = new Vector3(-1, -1, 0);
userPrimitives[2].TextureCoordinate = new Vector2(0, 1);
userPrimitives[3] = new VertexPositionTexture();
userPrimitives[3].Position = new Vector3(1, -1, 0);
userPrimitives[3].TextureCoordinate = new Vector2(1, 1);
```

Instead of having to declare a color for each vertex, you are now specifying a `TextureCoordinate` for each vertex.

Note

Although you are defining the texture coordinates in code explicitly, this is mostly done within a 3D content creation package when an artist is creating the models for your game.

The final changes to the code need to occur in your custom shader. Add a new global variable for the texture sampler you will use in the shader. Add the following global variable to your custom shader effect file.

```
sampler ColorTextureSampler : register(s0);
```

`Sampler` is a special type defined in the HLSL language. It lets the compiler know that the variable is used as a texture sampler. The register definition at the end is not required but does enable you to specify which of the samplers the variable will map to. In general, the sampler variables map to the order they are declared. Explicitly defining which sampler to use enables you more control if you are using multiple textures and want to specify textures to map to specific samplers.

The vertex shader input and output structures need to be updated to include the texture coordinate to use per vertex.

```
struct VertexShaderInput
{
    float4 Position : POSITION0;
    float2 TexCoord : TEXCOORD0;
};

struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float2 TexCoord : TEXCOORD0;
};
```

The `TexCoord` field is passed in as part of the vertices that you defined in your quad. Notice that the `TEXCOORD0` semantic is used. Vertices can contain multiple texture coordinates, and this specifies that this field should be the first one defined in the `VertexDeclaration`.

The vertex shader then needs to copy this value directly to the `VertexShaderOutput`. Add the following line to your vertex shader:

```
output.TexCoord = input.TexCoord;
```

The final step is to update the pixel shader to read the final color from the texture sampler and to output the color. Replace your pixel shader with the following:

```
float4 Color = tex2D(ColorTextureSampler, input.TexCoord);
return Color;
```

Use the `tex2D` intrinsic function to return the color from the texture sampler. The input texture coordinate are interpolated across the face of the triangle using the rules setup in the `SamplerState`.

Running the sample code now should display a quad with the fractal texture and looks like Figure 8.5.

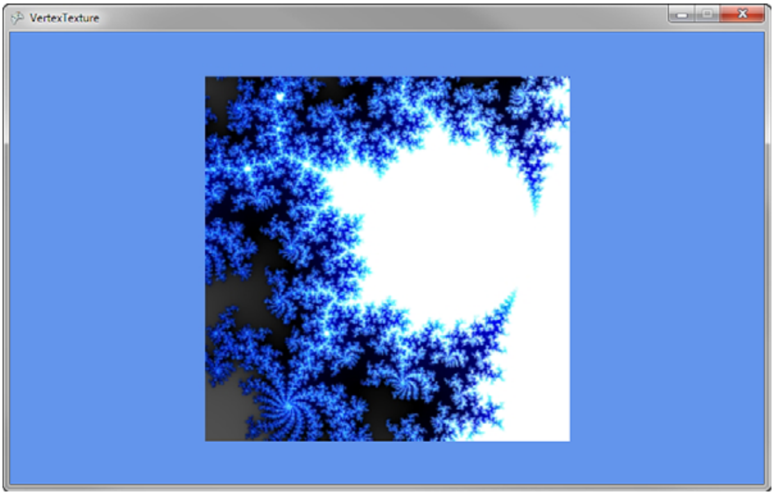


Figure 8.5 Drawing a textured quad

Setting Sampler States in Effect File

The preceding code used the XNA Framework APIs to set the sampler state and texture to use when running the effect. It is also possible to set this directly in the effect file itself so it does not require code changes within your project.

> **Note**
>
> Be mindful when setting states within the effect file. These changes can affect how other draw calls are rendered, so be watchful of which states you change.

To set the sampler state in the effect file, update the file with the following global texture and sampler:

```
texture ColorTexture;
sampler ColorTextureSampler : register(s0) = sampler_state
{
   Texture = (ColorTexture);
   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Wrap;
   AddressV  = Wrap;
};
```

You have added a new variable to hold the texture to use. The `ColorTextureSampler` has been updated to specify the sampler settings to use.

Because you added the new `ColorTexture` effect variable, you also need to set this in your game. In your game's `LoadContent` method, add the following line of code after you load the effect:

```
customEffect.Parameters["ColorTexture"].SetValue(colorTexture);
```

Instead of setting the texture on the `GraphicsDevice`, pass which texture to use directly to the effect file. When using sample states directly in the effect file, these settings take affect after you call `Apply` on the `EffectPass` that is using them.

Running the sample now should look just like before. Let's change that by showing off some of the sampler states.

Textures Repeating

In the previous examples, the texture coordinates ranged from 0 to 1, but you are not limited to just using these values. At times, it is useful for textures to repeat over and over across triangles. For example, you might make a sports game that has a large grass field. Creating a large texture that covers the whole field can be a large memory expense. An option is to have a smaller texture that repeats itself over the field every few feet. The higher number used for the texture coordinates, the more number of times the texture repeats.

There are two ways for the texture to repeat itself. The first is called wrapping, which causes the texture to just start over at the end of the first texture. The second is called mirroring, which causes the texture to reverse each time it is repeated.

To see some examples of texture, update the vertices deceleration.

```
// Create the verticies for our triangle
userPrimitives = new VertexPositionTexture[4];
userPrimitives[0] = new VertexPositionTexture();
userPrimitives[0].Position = new Vector3(-1, 1, 0);
userPrimitives[0].TextureCoordinate = new Vector2(0, 0);
userPrimitives[1] = new VertexPositionTexture();
userPrimitives[1].Position = new Vector3(1, 1, 0);
userPrimitives[1].TextureCoordinate = new Vector2(2, 0);
userPrimitives[2] = new VertexPositionTexture();
userPrimitives[2].Position = new Vector3(-1, -1, 0);
userPrimitives[2].TextureCoordinate = new Vector2(0, 3);
userPrimitives[3] = new VertexPositionTexture();
userPrimitives[3].Position = new Vector3(1, -1, 0);
userPrimitives[3].TextureCoordinate = new Vector2(2, 3);
```

The important changes are the `TextureCoordinate` values. Originally, the coordinates were set between 0 and 1 for both the `U` and `V` coordinates. You updated the code to set the max `U` value to 2 and the max `V` value to 3. This causes the resulting image to repeat two times from left to right and three from top to bottom.

Running the code now produces results similar to Figure 8.6. Notice that the texture wraps two times from left to right and three times from top to bottom.

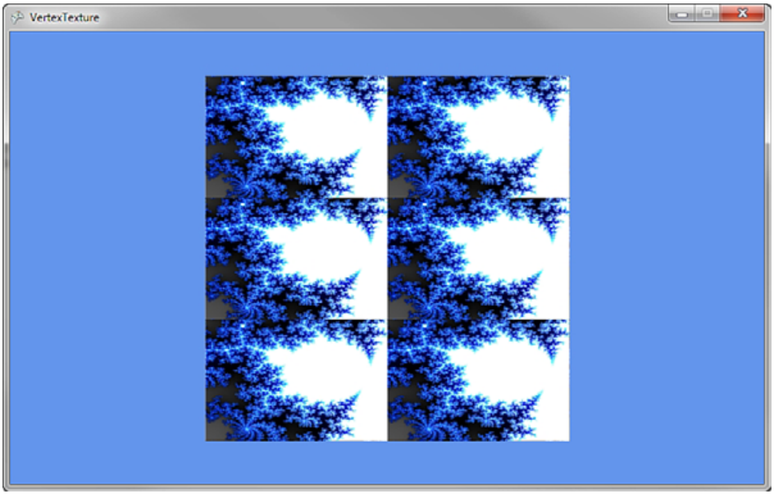


Figure 8.6 Drawing a textured quad with texture wrapping

The texture sampler uses wrapping because you set that in the effect file. If you update the sampler to use mirroring, you see different results.

Update the sampler in the effect file to use the following code:

```
sampler ColorTextureSampler : register(s0) = sampler_state
{
   Texture = (ColorTexture);
   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Mirror;
   AddressV  = Mirror;
};
```

You changed both the `AddressU` and `AddressV` modes to `Mirror`. The texture now displays in reverse on every other repeat of the texture.

Running the code now produces a mirroring effect and looks like Figure 8.7.

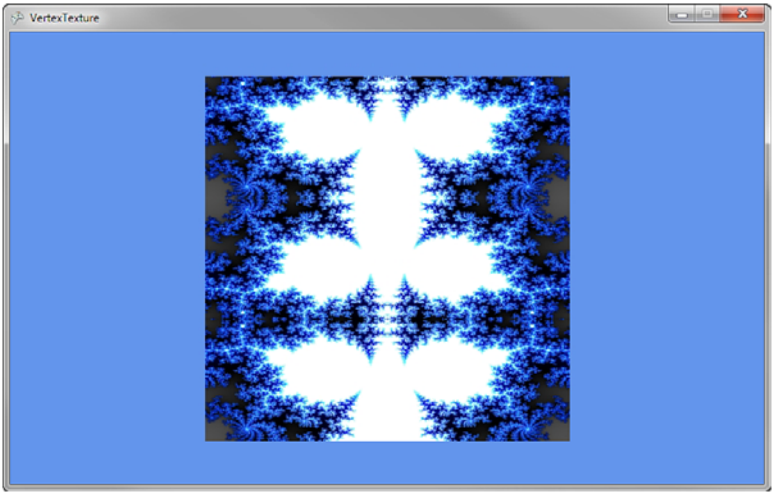


Figure 8.7 Drawing a textured quad with texture mirroring

Lighting

To produce more realistic 3D objects, you need to add simulated lighting and shading to the objects you are drawing. There are many different types of lighting models that simulate how light works in the real world. Simulated lighting models have to strike a balance between realism and performance. Many of the lighting models used within 3D graphics are not based on how light physically works in the real world; instead, the models try to simulate how the light looks reflected off different types of objects.

We look at some of the common lighting models that are used in 3D graphics including those used in the built-in `BasicEffect`.

Ambient Lighting

The simple light to simulate is light that has no general direction and has a constant intensity in all directions on an object. This light is scattered many times meaning it has bounced off many other objects before hitting the final object you are shading.

In the real world, this type of light occurs when you are outside but in the shade. Although you are not in the direct sunlight, there is plenty of light that bounces around off other objects to light you and your sounding objects.

In 3D graphics, ambient lighting is used to give the lowest possible level of lighting an object can have in your scene when it is not lit by other types of lighting. The light value is represented by a `Vector3` that describes the color in three colors: red, green, and blue using the `X`, `Y`, and `Z` properties. A value of 0 means no light, and a value of 1 means it is fully lit, which is not likely for your ambient light.

To see how ambient lighting looks when used in a game, let's create a sample that displays four objects: a cylinder, a sphere, a torus, and a flat plane on the ground. You use these objects through your lighting examples while you continually add different types of lighting to the example.

The first step is to add the following member variables to your game class:

```
Effect ambientEffect;
Model model;
Matrix[] modelTransforms;

Matrix world;
Matrix view;
Matrix projection;

// This is both the light color and the light intensity
Vector3 ambientLightColor;
// The color of the objects you are going to draw
Vector3[] diffuseColor;
```

The `ambientEffect` variable stores your custom effect. Update the name of the effect variable in each of the examples with the type of lighting you are using. The next two variables, `model` and `modelTransforms`, are used to store the `Model` and transform hierarchy for the `ModelMesh`'s contained within the `Model`.

The next three `Matrix` values are used to store the matrices to transform the vertices of the `Model` from local space into screen projection space.

The final two variables are used for the lighting calculations. The `ambientLightColor` represents the light color and intensity of the ambient light. The `diffuseColor` array contains the colors for the three objects you plan to draw. This is the color of the object if it was fully lit. Unlike the vertex color example, you set the color to draw the objects by passing the value into the shader as a global constant. If the model contained vertex colors, use those values instead.

Next, in the game's `Initialize` method, set the initial values for the previous variables.

```
// Set the values for our world, view, and projection matrices
world = Matrix.Identity;
view = Matrix.CreateLookAt(new Vector3(0, 1.5f, 3.5f), new Vector3(0, 0, 0), new
Vector3(0, 1, 0));
projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
GraphicsDevice.Viewport.AspectRatio, 0.1f, 100.0f);

// Set the ambient color
ambientLightColor = new Vector3(0.4f, 0.4f, 0.4f);

// Set the diffuse colors
diffuseColor = new Vector3[4];
diffuseColor[0] = new Vector3(1, 0.25f, 0.25f);
diffuseColor[1] = new Vector3(0.25f, 1, 0.25f);
```

```
diffuseColor[2] = new Vector3(0.25f, 0.25f, 1);
diffuseColor[3] = new Vector3(0.5f, 0.0f, 0.5f);
```

First, set the matrices to sensible values to frame the models close to the screen. Then, set the color of the ambient light. Use values of 0.4f or 40 percent. This value is combined with each of the color values set in the next array. Set red, green, blue, and purple values for each of the objects.

Next, load the custom effect file and the model file. In the game's `LoadContent` method, add the following lines of code:

```
// Load custom effect
ambientEffect = Content.Load<Effect>("AmbientLighting");

// Load our model
model = Content.Load<Model>("LightingModels");
// Create an array for model transforms and make a copy
modelTransforms = new Matrix[model.Bones.Count];
model.CopyAbsoluteBoneTransformsTo(modelTransforms);
```

Most of the code should look familiar. You load the `Effect` file that you will create shortly and a `Model` that contains the four objects that will display the lighting models. The final two lines of code populate a `Matrix` array with the local `Matrix` transforms for each of the `MeshParts`. Because the fbx file contains multiple objects, they are represented as separate `MeshParts`, each with its own local `Matrix` that is used to transform the object into world space.

The final piece of your game code is to update the game's `Draw` method with the following code:

```
int diffuseIndex = 0;


// Set effect wide parameters
ambientEffect.Parameters["View"].SetValue(view);
ambientEffect.Parameters["Projection"].SetValue(projection);
ambientEffect.Parameters["AmbientLightColor"].SetValue(ambientLightColor);

foreach (ModelMesh mesh in model.Meshes)
{
    // Set mesh effect parameters

ambientEffect.Parameters["World"].SetValue(modelTransforms[mesh.ParentBone.Index] *
                                                   world);

    foreach (ModelMeshPart meshPart in mesh.MeshParts)
    {
        // Set vertex and index buffer to use
        GraphicsDevice.SetVertexBuffer(meshPart.VertexBuffer,
➥meshPart.VertexOffset);
        GraphicsDevice.Indices = meshPart.IndexBuffer;
        // Set diffuse color for the object

ambientEffect.Parameters["DiffuseColor"].SetValue(diffuseColor[diffuseIndex++]);

        // Apply our one and only pass
        ambientEffect.CurrentTechnique.Passes[0].Apply();

        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                                                meshPart.NumVertices,
                                                meshPart.StartIndex,
                                                meshPart.PrimitiveCount);
    }
}
```

Let's break the code down.

The first variable `diffuseIndex` is used to offset into the color array to set the color of each of the `MeshParts` in the `Model`.

The next section sets the `Effect` parameters that don't change over the course of drawing the different parts of the `Model`. In this case, you set the `View` and `Projection` matrix along with the `AmbientLightColor`. Because setting effect parameters can be expensive, set only new values when you have to. In this case, set them at the highest level and once per use of the `Effect`.

The next section of code loops over all of the `ModelMeshs` in the `Model`. The `World` matrix is then set by combining the local transform from the `ModelMesh` with the `world` matrix for the scene.

The final section of code loops over all the `ModelMeshParts` within each of the `ModelMeshs`. Set the `VertexBuffer` and `IndexBuffer`, and then set `Effect` parameters that change per object you draw. In this case, set the color for each object before you call `EffectPass.Apply`. Each mesh is then drawn by calling `DrawIndexedPrimitives`.

You have all of the C# game code you need to draw your model using ambient lighting. Now you need to create the custom effect that you need to use with the previous bit of code. Create a new effect file as you have done previously in this chapter and name it `AmbientLighting.fx`.

Along with the template global variables, add the following global variables:

```
float3 AmbientLightColor;
float3 DiffuseColor;
```

These store the ambient light and object color.

Leave the `VertexShaderInput` and `VertexShaderOutput` structures as is from the template along with the vertex shader `VertexShaderFunction`.

Finally, update the pixel shader with the following code:

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
    // Store the final color of the pixel
    float3 finalColor = float3(0, 0, 0);
    // Add in ambient color value
    finalColor += AmbientLightColor * DiffuseColor;

    return float4(finalColor, 1);
}
```

This is a simple pixel shader. It starts by declaring the variable that is used to store the return final color of the pixel. The ambient lighting is then calculated by multiplying the ambient light by the color of the object. The final color is then returned adding the fourth channel for alpha with a value of 1 for fully opaque.

Running this sample should display something similar to Figure 8.8.

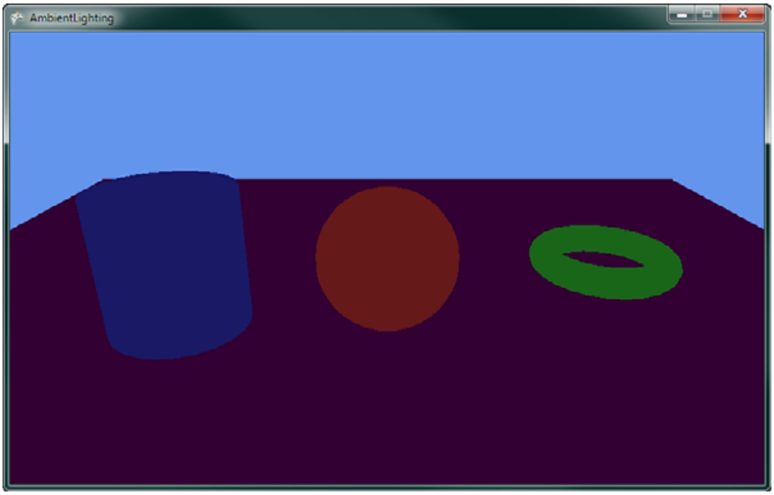


Figure 8.8 Ambient lighting

Notice how each of the models is a constant color across the entire mesh. The direction of the triangle is not taken into account with this ambient lighting model.

Triangle Normals

For more realism, take into account the direction the triangle faces in regards to the light. To help determine the direction a triangle is facing, use the normal of the triangle. The normal contains only a direction and, therefore, should always have a length of 1. It is important that you normalize or set the length to 1 of a normal anytime you perform a calculation that alters the normal's size.

There are two types of normals when working with a triangle. The first is called a face normal and is defined to be perpendicular to the plane that is defined by the three points that make up the triangle. Figure 8.9 shows an example of a face normal.

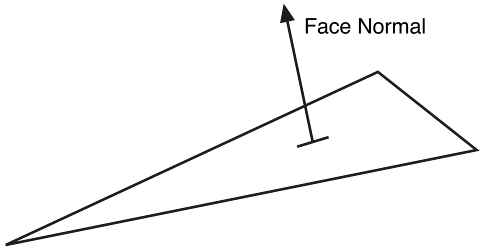


Figure 8.9 Face normal

In real-time 3D graphics, the second type of normal called a vertex normal is used. Each vertex of the triangle defines its own normal. This is useful because you might want the object to appear smooth. In this case, the normal at a vertex is averaged with values from adjacent triangles to enable a smooth transition from one to the other. Figure 8.10 shows an example of vertex normals.

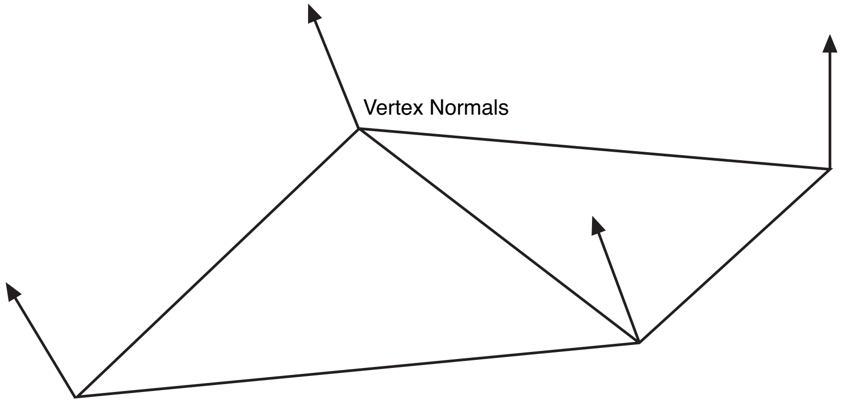


Figure 8.10 Vertex normals

You can update the previous example to display the normal values of the mesh with just a few code changes. No changes need to be made on the game code side, and you need to make only a couple of changes to the shader.

Update the input and output vertex structures to the following:

```
struct VertexShaderInput
{
    float4 Position : POSITION0;
    float3 Normal   : NORMAL;
};
```

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float3 Normal   : TEXCOORD0;
};
```

The Normal value is added to each structure. The NORMAL semantic used in the input structure tells the graphics card that you want the normal data from the model. It is matched to the corresponding data from the `VertexBuffer` where the `VertexDeceleration` has set the normal channel.

> **Note**
>
> The model used in this example contains normal data. This exports from the modeling package used to create the model. If your model does not contain normal data, then you see an exception when you try to draw the model.

In the vertex shader, pass the normal data from the input structure to the output structure. Add the following line of code before you return the output structure:

```
output.Normal = input.Normal;
```

The normal data interpolates between each vertex across the triangle for each pixel. In the pixel shader, read this normal value and use the components of the vector as the red, green, and blue color.

Update the pixel shader with the following line of code that returns the normal data as a color:

```
return float4(normalize(input.Normal), 1);
```

The normal needs to be normalized because the interpolation can lead to normals with length not equal to 1. The three components of the normal are then combined with an alpha value of 1 to color the pixel. If you run the example, it displays a rainbow of colors similar to Figure 8.11.

Diffuse Lighting

The term diffuse means to scatter or become scattered, so the diffuse light is reflected light that bounces off in many directions causing an object to appear to be flat shaded and not shinny. Ambient lighting, which gives a constant color across the triangles in a mesh diffuse lighting, differs depending on the angle of the triangle to the light source. Use Lambert's cosine law, which is a common equation used to determine the diffuse color. This law states that the light reflected is proportional to the cosine of the angle between the normal and the light direction.

The type of lighting you are going to model first is called directional light. The light is considered to come from a constant direction in parallel beams of light. This is similar to how sunlight reaches earth.

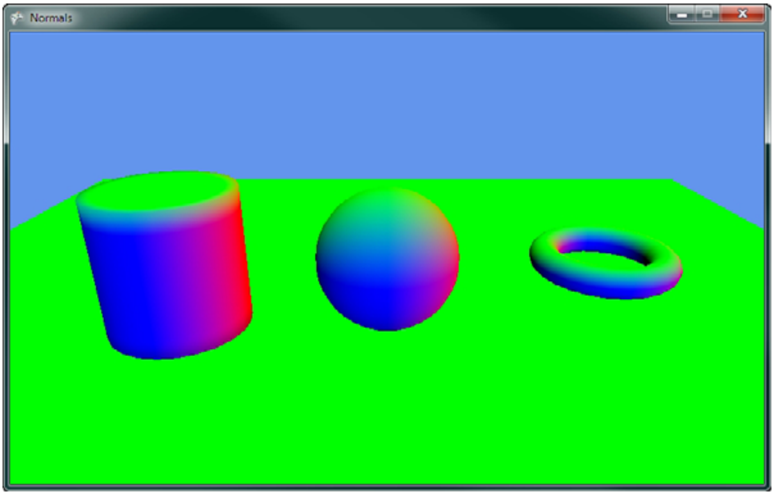


Figure 8.11 Geometry colored by their normal values

Note

Sunlight is not parallel but for 3D graphics purposes, it can be treated that way because the size and distance of the sun is so great that the light appears to reach earth as parallel beams.

Because Lambert says you can use the cosine of the angle between the normal and the light, you can easily calculate this by taking the dot product of the normal and the light direction vectors. If both are unit length, then the dot product is equal to the cosine of the angle, which is the value you want.

Figure 8.12 shows the directional lights parallel rays hitting the triangle normals and the angle calculation.

Let's add some diffuse lighting from a directional light to the previous example of ambient lighting. The first thing you need are some additional member variables in your game class.

```
// The direction the light comes from
Vector3 lightDirection;
// The color and intensity of the diffuse light
Vector3 diffuseLightColor;
```

The first variable `lightDirection` is exactly what the name describes—the direction the light is going in. There are two ways to describe the direction of a directional light. The first is to describe the direction the light is moving in. This is the way we describe the light in the example. The second is to describe the direction to the source of the light like pointing towards the sun. This is the way you need the value in your shader so you can perform the angle calculation using the dot product.

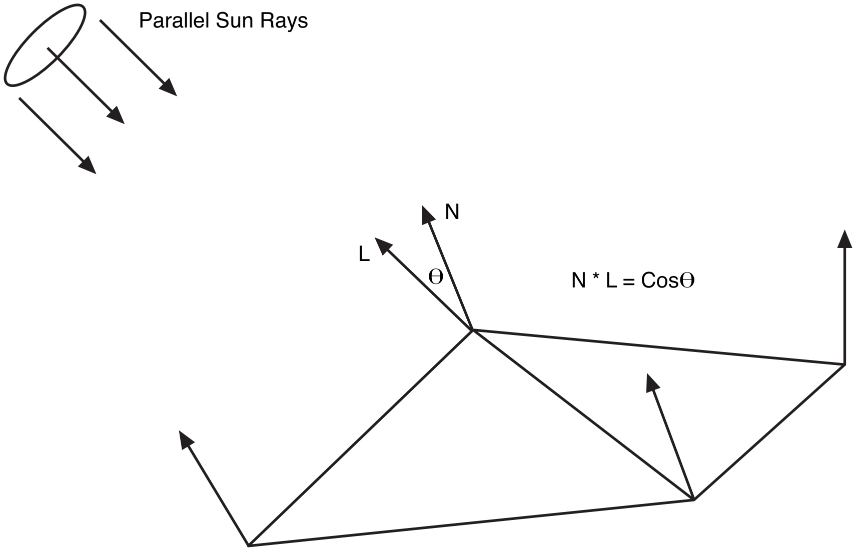


Figure 8.12 Directional light hitting triangle

The second variable is the color of the light. Lights don't always have to be white; they can be different colors. Each color channel affects the same color channel of the object's diffuse color.

In your game's `Initialize` method, add the following lines of code to set the light's direction and color. Note that the direction is normalized to keep the vector at unit length.

```
// Set light starting location
lightDirection = new Vector3(-0.5f, -0.5f, -0.6f);
lightDirection.Normalize();

// Set the lights diffuse color
diffuseLightColor = new Vector3(1, 0.9f, 0.8f);
```

The final changes you need to make to your game code is to send the values to the Effect. Set the `LightDirection` and `DiffuseLightColor` just after the other effect wide parameters as the following code shows. The light direction is negated to change it from pointing from the light to be the direction to the light. This is the format you need in your shader, so make it here instead of calculating the negation multiple times in the shader.

```
// Set effect wide parameters
diffuseEffect.Parameters["View"].SetValue(view);
diffuseEffect.Parameters["Projection"].SetValue(projection);
diffuseEffect.Parameters["AmbientLightColor"].SetValue(ambientLightColor);
diffuseEffect.Parameters["LightDirection"].SetValue(-lightDirection);
diffuseEffect.Parameters["DiffuseLightColor"].SetValue(diffuseLightColor);
```

Now, update your custom effect file to calculate the diffuse color in addition to the ambient color.

The first change is to add two new global variables that are used to store the light direction and color.

```
float3 LightDirection;
float3 DiffuseLightColor;
```

Like the normal example, add the vertex normal to both the input and output vertex structures.

```
struct VertexShaderInput
{
    float4 Position : POSITION0;
    float3 Normal   : NORMAL;
};

struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float3 Normal   : TEXCOORD0;
};
```

The normal values also need to be passed from the input to the output structure in the vertex shader.

```
output.Normal = mul(input.Normal, World);
```

Finally, update the pixel shader to calculate the diffuse color and output the color for the pixel. Update the pixel shader with the following code:

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
    // Normalize the interpolated normal
    float3 normal = normalize(input.Normal);

    // Store the final color of the pixel
    float3 finalColor = float3(0, 0, 0);

    // Start with ambient light color
    float3 diffuse = AmbientLightColor;

    // Calculate diffuse lighting
    float NdotL = saturate(dot(normal, LightDirection));
    diffuse += NdotL * DiffuseLightColor;

    // Add in diffuse color value
    finalColor +=  DiffuseColor * diffuse;

    return float4(finalColor, 1);
}
```

First, the pixel shader normalizes the input normal. You need to normalize this value because the interpolation between vertices can lead to the vector not having a unit length. Then, set the minimum value for the diffuse lighting to the ambient light value. This is the minimum that the pixel can be lit. The additional light from the directional light is added to this value.

To calculate the light from the directional light, calculate the value of the dot product of the normal and the light direction. Use the `saturate` intrinsic function to clamp the value between 0 and 1. If the dot product is negative, then it means the normal is facing away from the light and should not be shaded so you want a value of 0 and not the negative value of the dot product.

The `NdotL` value is then multiplied by the color of the directional light and added to the diffuse light amount. The diffuse light amount is then multiplied by the diffuse color of the object itself to obtain the final color of the object. The final color is then returned with an alpha value of 1.

If you run the previous code sample, you should see something similar to Figure 8.13.

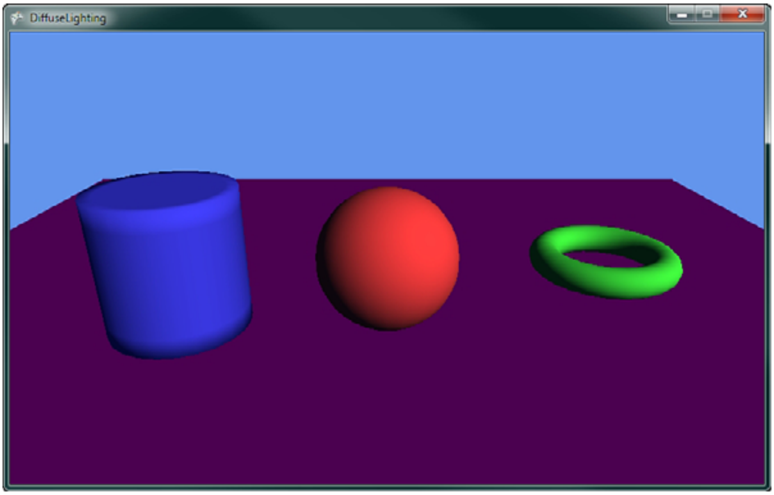


Figure 8.13 Directional diffuse lighting

Multiple Lights

In the real world, you have more than one light source. You can also have more than one directional light. To add an additional light, add an additional light direction and color to your game class.

```
// The direction and color of a 2nd light
Vector3 lightDirection2;
Vector3 diffuseLightColor2;
```

Next, give them some default values in the game's `Initialize` method.

```
// Set the 2nd lights direction and color
lightDirection2 = new Vector3(0.45f, -0.8f, 0.45f);
lightDirection2.Normalize();
diffuseLightColor2 = new Vector3(0.4f, 0.35f, 0.4f);
```

Then, send the values to the Effect.

```
diffuseEffect.Parameters["LightDirection2"].SetValue(-lightDirection2);
diffuseEffect.Parameters["DiffuseLightColor2"].SetValue(diffuseLightColor2);
```

In the shader effect file, add the following two new global variables:

```
float3 LightDirection2;
float3 DiffuseLightColor2;
```

In the pixel shader, calculate the dot product of the additional light and add the value to your diffuse light value before adding the value to the `finalColor`.

```
// Calculate 2nd diffuse light
NdotL = saturate(dot(normal, LightDirection2));
diffuse += NdotL * DiffuseLightColor2;
```

Running the example now should show something similar to Figure 8.14.

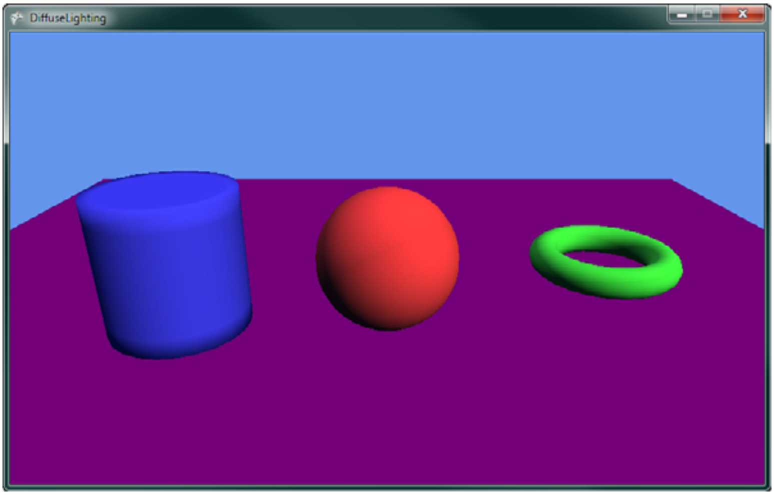


Figure 8.14 Directional diffuse lighting from two light sources

Oversaturation

As you add more lighting, the possibility of oversaturation becomes a concern. Notice that lighting is additive. As you add more lights, the final color channel values can go above 1, which is full color. As the values go above 1, no change in the final pixel color output the screen occurs. Differences of colors above 1 appear to be the same color to the user. Portions of an object might lose their definition becoming bright white or another solid color. You can limit oversaturation by lowering the amount of lights and keeping the color intensity values of the lights lower. Notice that the previous example used smaller values for the second light's color. Often, you have one stronger light with an additional couple of weaker lights.

Emissive Lighting

Some objects not only receive light but also give off light. Emissive light is the light given off by the object. The emissive light is added to other light sources.

To add emissive light to the continuing example is quite simple. You need one additional color value for each object you want to draw. Add the following to your game class member variables:

```
// The emissive color of the objects
Vector3[] emissiveColor;
```

You will store a separate emissive color for each object like you did for the diffuse color. In the game's `Initialize` method, add the following lines of code:

```
// Set the emissive colors
emissiveColor = new Vector3[4];
emissiveColor[0] = new Vector3(0, 0.75f, 0);
emissiveColor[1] = new Vector3(0, 0, 0.75f);
emissiveColor[2] = new Vector3(0.75f, 0, 0);
emissiveColor[3] = new Vector3(0, 0.75f, 0);
```

Next, pass this color value into your effect. Just after you set the `DiffuseColor`, set the `EmissiveColor` for the effect.

```
// Set diffuse color for the object
emissiveEffect.Parameters["DiffuseColor"].SetValue(diffuseColor[colorIndex]);
emissiveEffect.Parameters["EmissiveColor"].SetValue(emissiveColor[colorIndex]);
```

In the effect file, you need an additional global variable that will store the emissive color.

```
float3 EmissiveColor;
```

Finally, in the pixel shader, add the emissive color before you return the `finalColor`.

```
// Add in emissive color
finalColor += EmissiveColor;
```

Running the example now shows the objects with an inner glow of the emissive colors like Figure 8.15.

Although the objects look like they are emitting light, they don't have any effect on other objects. This emissive light is used only on the object itself and does not create a halo around the object.

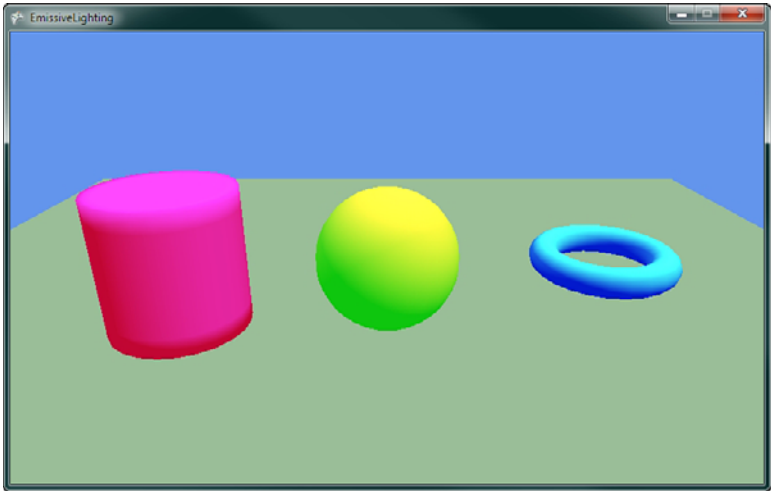


Figure 8.15 Emissive lighting on the objects

For the remainder of the chapter, turn off the emissive lighting by setting the emissive color to zero.

Specular Lighting

In the real world, not all objects are flat shaded. Some objects are shinny and reflect light very well. Think of a bowling ball and how shinny it is. If you look at a bowling ball, notice how there might be bright spots on the ball where the lights in the bowling alley are better reflected.

The shinny spots appear where the angle of the reflected angle from the light about the vertex normal is close to the vector to the camera position. Figure 8.16 shows how the specular light on the triangle is dependent on the viewer angle.

Phong Shading

There are a number of ways to model this shinny appearance of objects. One method is called Phong shading, which is named after its inventor Bui Tuong Phong.

Phong shading uses two new vectors `R` and `V`. `R` is the unit reflection vector of the light about the vertex normal. `V` is the unit vector of the camera position to the vertex rendered called the viewer vector. The intensity of the specular highlight is then calculated by taking the dot product between `R` and `V`. Different materials have different levels of shininess to achieve different results a specular power values are used to raise the `R dot V` to different powers. This calculated specular intensity value is then multiplied by the object's specular color and then added to the final color of the pixel.

The equation for the specular shading value using the Phong lighting model is

$$\text{Specular} = (R \bullet V)^{\text{SpecularPower}} \star (\text{Light Color}) \star (\text{Object Color})$$

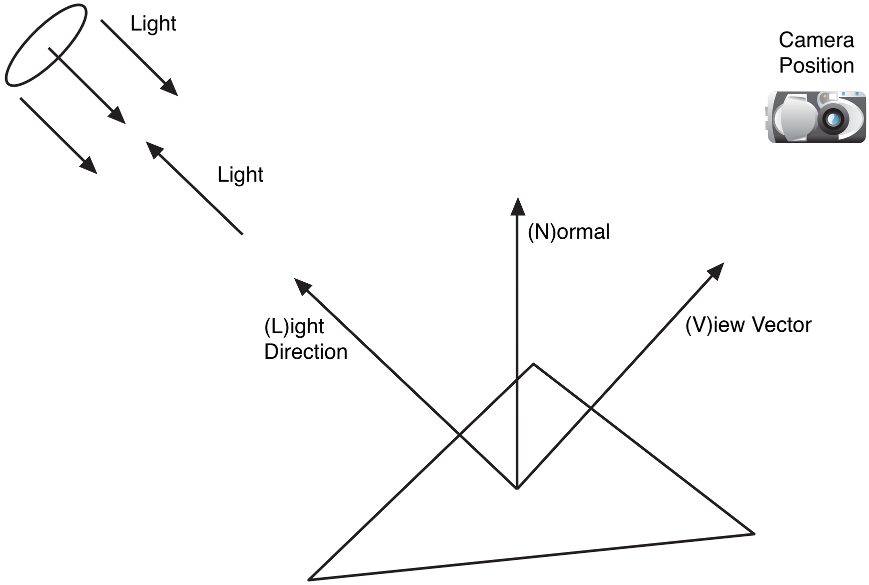


Figure 8.16 Specular light on a triangle

Blinn-Phong Shading

Calculating the reflection vector requires a number of calculations. To eliminate some of these calculations, Jim Blinn created another specular lighting model based on Phong, called Blinn-Phong in 1977.

Blinn-Phong differs from Phong by using a new vector `H` called the half vector. The half vector is the vector halfway between the viewer vector `V` and the light direction `L`. The half vector can be calculated by adding the view vector `V` and the light direction `L` and normalizing the vector to unit length. The `H` vector is dot multiplied with the vertex normal and raised to a specular power. This is similar to how the `R` and `V` vectors are used in the Phong lighting model.

The equation for the specular shading value using the Blinn-Phong lighting model is

$$\text{Specular} = (H \bullet N)^{\text{SpecularPower}} \star (\text{Light Color}) \star (\text{Object Color})$$

Let's add some Blinn-Phong to the previous example. Add the following member variables to your game:

```
// The specular color of the objects
// The w component stores the specular power
Vector4[] specularColorPower;
// Specular color of the light
Vector3 specularLightColor;
// The position of the camera
Vector3 cameraPosition;
```

The first array `specularColorPower` stores the specular color and specular power for each of the objects. The color uses the `X`, `Y`, and `Z` components of the vector while the `W` component stores the specular power. The `specularLightColor` variable stores the specular color of the light source. The final value `cameraPosition` stores the camera location.

In the game's `Initialize` method, add the following values to set the specular color of the objects and the light. Also set the camera position that you used to make the view matrix.

```
// Set the specular color and power
specularColorPower = new Vector4[4];
specularColorPower[0] = new Vector4(1, 1, 1, 32.0f);
specularColorPower[1] = new Vector4(1, 1, 0, 64.0f);
specularColorPower[2] = new Vector4(0, 1, 1, 32.0f);
specularColorPower[3] = new Vector4(0, 0, 0, 0);

// Set the lights specular color
specularLightColor = new Vector3(1, 0.9f, 0.8f);

// We set the camera position
cameraPosition = new Vector3(0, 1.5f, 3.5f);
```

These values need to be set on the effect. The `SpecularLightColor` and `CameraPosition` can be set with other effect wide properties.

```
specularEffect.Parameters["SpecularLightColor"].SetValue(specularLightColor);
specularEffect.Parameters["CameraPosition"].SetValue(cameraPosition);
```

The `SpecularColorPower` needs to be set with other per object effect values such as the diffuse color.

```
specularEffect.Parameters["SpecularColorPower"].SetValue(specularColorPower[color
➥Index]);
```

That is it for the game code changes. Now, you need to update the effect file to add the Blinn-Phong calculations.

First, add some additional global variables to your effect.

```
float4 SpecularColorPower;
float3 SpecularLightColor;
float3 CameraPosition;
```

The next change is to the output vertex structure where you add the view vector `V`, which you calculate in the vertex shader.

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float3 Normal   : TEXCOORD0;
    float3 View     : TEXCOORD1;
};
```

In the vertex shader, the `View` value is calculated by subtracting the calculated world position from the camera position, which is also in world space.

```
output.View = CameraPosition - worldPosition;
```

The final change updates the pixel shader to calculate the specular lighting value and adds it to the final pixel color.

```
// Normalize the interpolated view vector
float3 view = normalize(input.View);

// Calculate half vector
float3 half = normalize(view + LightDirection);

// Calculate N * H
float NdotH = saturate(dot(normal, half));

// Calculate specular using Blinn-Phong
float specular = 0;
if (NdotL != 0)
    specular += pow(NdotH, SpecularColorPower.w) * SpecularLightColor;

// Add in specular color value
finalColor +=  SpecularColorPower.xyz * specular;
```

The first line normalizes the `View` vector, which needs to be unit length and can change as it is interpolated across the triangle. The next line calculates the half vector by adding the view and light direction vectors and then normalizes the result. The dot product of `N` and `H` are then taken and clamped between 0 and 1. Finally, its specular value is calculated by using the `pow` intrinsic function that raises the `NdotH` value to the specular power, which passed in as the `w` component of the `SpecularColorPower` variable and is then multiplied by the light's specular color.

The last bit of code adds the specular color to the final pixel color using the calculated secular intensity and the object's specular color stored in the `xyz` channels of `SpecularColorPower`.

Running the sample now should produce results with a shinny spot on each of the objects in the scene as shown in Figure 8.17.

Try adjusting all of the lighting and color values in the examples thus far and see how they change the results. Notice that lowering the specular power makes the specular highlight larger but less crisp around its edges.

Fog

Real-world objects that are farther away are not only smaller but also obscured by the amount of atmosphere between the viewer and the object. On most days when the air is clear, the visibility of objects far away are not obscured too much, but on other days a layer of fog might cover the ground lowering the visibility.

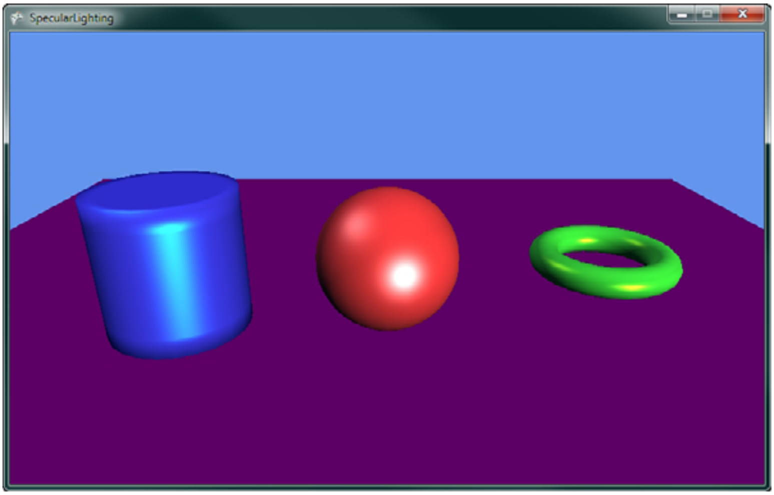


Figure 8.17 Blinn-Phong specular lighting on objects

In games, you can simulate the fog that obscures objects by interpolating the color of an object as it moves farther into the distance with a fog color. Fog can be any color, but all objects should use a similar fog color; otherwise, some objects will stand out more than others.

There are multiple ways to interpolate the fog value as distance increases. The first method is to use a simple linear interpolation as distance increases. The equation you will use to calculate the linear fog value is

Fog = (Distance – Fog Start) / (Fog End – Fog Start)

The distance is from the camera to the current pixel you are drawing. The fog start is the distance that the fog can start to be seen or where the interpolation should start. The fog end is the distance where the object will be in complete fog. The calculated fog is a floating point value, which is in the range of 0 and 1. Values over 1 should be treated as 1, which means fully in fog. The fog value is then used as the interpolation value of how much of the fog color to use with the final calculated pixel color.

To add linear fog to the ongoing sample, you will add three new member variables to your game.

```
// Fog color and properties
Vector3 fogColor;
float fogStart;
float fogEnd;
```

These values store the fog color and distance values you need to send to your custom effect.

These values need to be set to some defaults, so in the game's `Initialize` method, add the following lines of code:

```
// Set fog properties
fogColor = Color.CornflowerBlue.ToVector3();
fogStart = 2;
fogEnd = 18;
```

Use a color value that is the same as your clear color, so that the objects appear to fade away as they move into the distance. In your game, set the color based on the time of day. If your game is at night, fade to a dark almost black color. If it is daytime, then use a lighter gray color. The fog color can also be used for special effects to simulate some type of toxic gas in a city so you can use some other interesting colors.

The other two values are set to sensible values for the scene. Depending on the size and depth of your scene and the camera's view, update these values to match the scales.

In the game's `Draw` method, send the new values to your custom effect.

```
fogEffect.Parameters["FogColor"].SetValue(fogColor);
fogEffect.Parameters["FogStart"].SetValue(fogStart);
fogEffect.Parameters["FogEnd"].SetValue(fogEnd);
```

Finally, update how you are drawing the meshes in sample so that you have many more objects that move farther and farther away from the camera.

```
// Draw the models 5 times in the negative Z direction
for (int i = 0; i < 5; i++)
{
    world = Matrix.CreateTranslation(0, 0, i * -4.0f);
    colorIndex = 0;

    foreach (ModelMesh mesh in model.Meshes)
    {
        // Set mesh effect parameters

fogEffect.Parameters["World"].SetValue(modelTransforms[mesh.ParentBone.Index]
                                               * world);

        foreach (ModelMeshPart meshPart in mesh.MeshParts)
        {
            // Set vertex and index buffer to use
            GraphicsDevice.SetVertexBuffer(meshPart.VertexBuffer,
➥meshPart.VertexOffset);
            GraphicsDevice.Indices = meshPart.IndexBuffer;

            // Set diffuse color for the object
            fogEffect.Parameters["DiffuseColor"].
            SetValue(diffuseColor[colorIndex]);
            fogEffect.Parameters["SpecularColorPower"].SetValue(
                                        specularColorPower[colorIndex]);

fogEffect.Parameters["EmissiveColor"].SetValue(emissiveColor[colorIndex]);
            colorIndex++;

            // Apply our one and only pass
            fogEffect.CurrentTechnique.Passes[0].Apply();

            GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                                                  meshPart.NumVertices,
                                                  meshPart.StartIndex,
                                                  meshPart.PrimitiveCount);
        }
    }
}
```

Loop five times to draw the model multiple times and farther and farther depths. This should enable you to get a better idea of the effect of the fog as the distance from the camera increases.

With all of the game code changes complete, you just need to make a few simple changes to your custom effect file to support the linear fog. First, you need the global variables for the fog color and distance properties.

```
float3 FogColor;
float FogStart;
float FogEnd;
```

Next, you need to calculate the fog value per vertex, so you need to pass the fog value into the pixel shader. To do this, add the following value to your vertex output structure.

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float3 Normal   : TEXCOORD0;
    float3 View     : TEXCOORD1;
    float Fog       : TEXCOORD2;
};
```

In the vertex shader, calculate the fog value before outputting the vertex.

```
// Calculate fog value
output.Fog = saturate((length(CameraPosition - worldPosition) - FogStart) /
                      (FogEnd - FogStart));
```

Use the same equation described previously. In this code, the distance is calculated by subtracting the world position of the vertex from the camera position. Then, use the `length` intrinsic function to calculate the length of the resulting vector. The `saturate` intrinsic function is used to clamp the calculated for value from 0 to 1.

The final change is to update the pixel shader to use the newly calculated fog value. Just before returning the final color from the pixel shader, add the following line of code:

```
// lerp between the computed final color and the fog color
finalColor = lerp(finalColor, FogColor, input.Fog);
```

The `lerp` intrinsic function interpolates between two vectors given the interpolation value, which should be between 0 and 1. Interpolate between the already calculated final color and the fog color depending on the fog value that you calculated in the vertex shader.

Running the sample now should display multiple versions of the models that are farther and farther away from the camera that slowly fade to the background color. This should look similar to Figure 8.18.

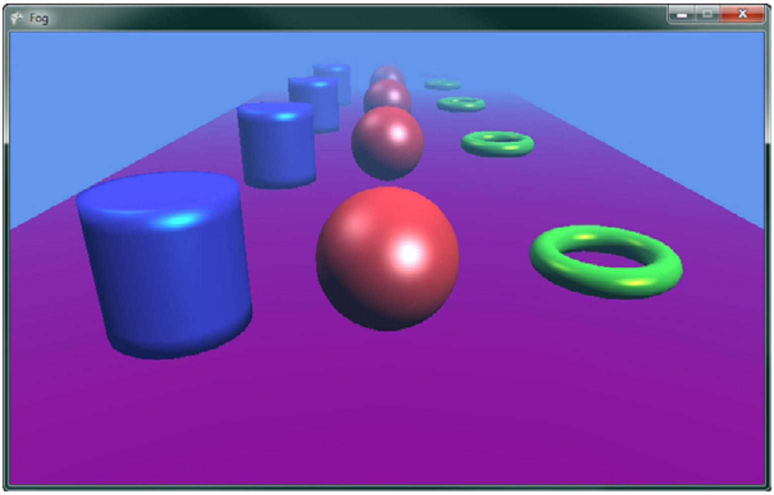


Figure 8.18 Objects obscured by fog as they move farther into the distance

The fog value does not have to interpolate linearly with distance. You can also use other types of interpolation such as an exponential falloff to achieve difference results. Because you have the control in the vertex shader, there are many different ways that you can use the distance to calculate the fog value.

Point Lights

All of the examples up to this point use one or more directional lights when calculating the light's influence on the color of the triangles. Although directional lights work well to simulate the type of light that is coming from distant objects such as the sun, they don't work to simulate how objects look indoors when lit from multiple smaller artificial lights such as light bulbs. In computer graphics, we call these types of light sources point lights.

Point lights have a known position in 3D space, which differs from directional lights that only have a direction. The direction to the point light must be calculated because it changes depending on the location of the object drawn.

The light that comes from smaller lights like the ones in your home tends to lose its brightness the farther away an object is from the light. You can see this in your home when you turn off the lights in a room and have only a single table lamp to light the room. Notice that the light is brightest close to the lamp, but falls off quickly with distance from the light. This falloff from the light source is called attenuation and differs depending on the size and type of light.

There are multiple ways in which you can calculate the attenuation when simulating the light that comes from point light sources. Use the following attenuation equation:

> Attenuation = 1 - (((Light Position - Object Position) / Light Range) * ((Light Position - Object Position) / Light Range))

In the previous equation, the light position is the location of the point light. The object position is the location you are currently drawing in world space, and the light range is how far the light rays will travel from the light source.

To demonstrate how point lights work in the game, update the previous specular lighting example, which uses directional lights.

First, you need some different member variables for the game. Remove the vector that represents the light direction and update it with the following code:

```
// The position the light
Vector3 lightPosition;
// The range of the light
float lightRange;
```

The light now has a position and a float value that represents the distance from the light that objects can be lit from.

In the game's `Initialize` method, give the point light the following values:

```
// Set light starting location
lightPosition = new Vector3(0, 2.5f, 2.0f);

// The range the light
lightRange = 6.0f;
```

In the same location where you set the light direction for the effect in your game's `Draw` method, update the effect with the new point light properties.

```
pointLightEffect.Parameters["LightPosition"].SetValue(lightPosition);
pointLightEffect.Parameters["LightRange"].SetValue(lightRange);
```

Most of the changes to support the point light occur in the effect file. You need two new global variables to store the light position and range.

```
float3 LightPosition;
float LightRange;
```

You need the position of the pixel you are rendering in the pixel shader in world space. To do this, add an additional value to the vertex output structure to store the position in world space.

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float3 Normal   : TEXCOORD0;
    float3 View     : TEXCOORD1;
    float3 WorldPos : TEXCOORD2;
};
```

The vertex shader needs to be updated to save the calculated world space position of the vertex.

```
output.WorldPos = worldPosition;
```

The pixel shader should be updated to the following:

```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
    // Normalize the interpolated normal and view
    float3 normal = normalize(input.Normal);
    float3 view = normalize(input.View);

    // Direction from pixel to the light
    float3 light = LightPosition - input.WorldPos;

    // Attenuation
    float attenuation = saturate(1 - dot(light / LightRange, light / LightRange));

    // Normalize the light direction
    light = normalize(light);

    // Store the final color of the pixel
    float3 finalColor = float3(0, 0, 0);

    // Start with ambient light color
    float3 diffuse = AmbientLightColor;

    // Calculate diffuse lighting
    float NdotL = saturate(dot(normal, light));
    diffuse += NdotL * DiffuseLightColor * attenuation;

    // Calculate half vector
    float3 half = normalize(view + light);

    // Calculate N * H
    float NdotH = saturate(dot(normal, half));
    // Calculate specular using Blinn-Phong
    float specular = 0;
    if (NdotL != 0)
        specular += pow(NdotH, SpecularColorPower.w) * SpecularLightColor *
➥attenuation;

    // Add in diffuse color value
    finalColor +=  DiffuseColor * diffuse;

    // Add in specular color value
    finalColor +=  SpecularColorPower.xyz * specular;

    // Add in emissive color
    finalColor += EmissiveColor;

    return float4(finalColor, 1);
}
```

Although it appears to be a lot of code at first, it is actually close to the pixel shader used for the specular lighting example. Let's walk though the important differences.

The input world position is interpolated for each pixel giving the location of the current pixel in world space, which is what you need to calculate the distance to the light source. The `light` variable is stored with the vector form the pixel to the light. The `attenuation` is then calculated using the equation from earlier in the chapter. The `light` vector is then normalized because you use this value as the direction of the light as you would if this was a directional light. The new normalized `light` is then used in the calculations of `NdotL` and `half`. Finally the `attenuation` is multiplied when calculating the `diffuse` and `specular` intensity values.

Running the sample now shows the objects in the scene lit from a point location where the light falls off the farther the objects are away from the light. Rotating the light or moving the light source from each frame can make this much more visible. Figure 8.19 shows the objects in the scene lit from a single point light.

Adding multiple point lights works in a similar way as adding additional directional lights. The lighting influence of each additional light source needs to be calculated for each light and added into the diffuse and specular intensity values. You can also mix and match by supporting a directional light and point lights in your effect.

Effect States

Chapter 7 discusses setting the different device states using the different state objects. Changing these states requires your game code to create different state objects and to set them on the device.

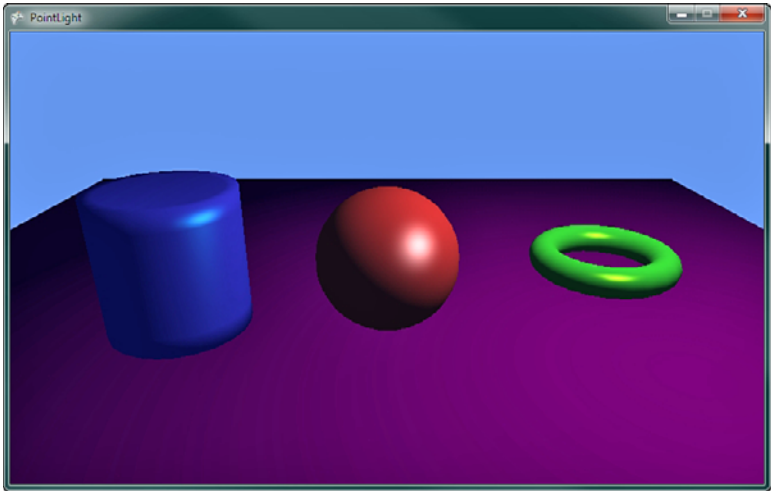


Figure 8.19 Objects lit by a point light source

Effect files have the capability to also set these states within their effect pass blocks. When your game calls `EffectPass.Apply`, if the effect pass contains any state changes, then they will be set on the device.

The default effect file template that is used when you create a new effect file in XNA Game Studio add a nice comment to let you know where you can set render states in the effect pass.

```
technique Technique1
{
    pass Pass1
    {
        // TODO: set renderstates here.

        VertexShader = compile vs_2_0 VertexShaderFunction();
        PixelShader = compile ps_2_0 PixelShaderFunction();
    }
}
```

There are many different states that can be set and the naming of the states and their values defers from the XNA Game Studio state objects. A listing of all of the states and values that are supported by an HLSL effect can be found at the following link:

http://msdn.microsoft.com/en-us/library/bb173347(VS.85).aspx

> **Note**
>
> XNA Game Studio limits some states and values that can be set in order to conform to the `Reach` and `HiDef` graphics profiles. This validation occurs when the effect file is built using the content pipeline. Reading any error from the build informs you of any states that are not supported in XNA Game Studio.

Alpha Blending Using Effect States

You can create an example of how to utilize the effect states by creating a custom effect that uses alpha blending to render the transparent objects. You utilize the existing specular sample that you created previously.

First, load a different model to draw. You use a model that has three separate objects that are close together so that when you make them transparent, you will see through each of them into the others. Update the `LoadContent` method with the following line of code. Also, add the model to your content project.

```
// Load our model
model = Content.Load<Model>("AlphaDemo");
```

To display the models with some transparency, update the diffuse colors to contain an extra alpha channel. The `diffuseColor` array needs to be updated to a `Vector4` from the current `Vector3`. Then in the `Initialize` method, update the `diffuseColor` array with the following values:

```
// Set the diffuse colors
diffuseColor = new Vector4[3];
diffuseColor[0] = new Vector4(1, 0.25f, 0.25f, 0.25f);
diffuseColor[1] = new Vector4(0.25f, 1, 0.25f, 0.5f);
diffuseColor[2] = new Vector4(0.25f, 0.25f, 1, 0.7f);
```

Unlink previous effects from this chapter—the effect you will create has multiple passes. This means that you will draw the geometry of the models multiple times, once for each `EffectPass`. Update the section of the `Draw` method, which calls `Apply` on the `EffectPass` and draws the primitives that make up the model with the following.

```
// Loop over all passes since we have more than one
foreach (EffectPass effectPass in
             alphaBlendingEffect.CurrentTechnique.Passes)
{
    effectPass.Apply();

    GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                                         meshPart.NumVertices,
                                         meshPart.StartIndex,
                                         meshPart.PrimitiveCount);
}
```

Because you have more than one pass, loop over all of the passes calling `Apply` and then drawing the indexed primitives that make up the `ModelMeshPart` you are currently drawing.

Now in the effect file, you need only a few small changes. The first is to update the pixel shader to use the `w` component of your diffuse color for the transparency value of the pixel.

```
// return the final color
return float4(finalColor, DiffuseColor.w);
```

Now, update the existing effect pass and add another.

```
technique Technique1
{
    pass Pass1
    {
        AlphaBlendEnable = true;
        SrcBlend         = SRCALPHA;
        DestBlend        = INVSRCALPHA;
        ZWriteEnable     = false;
        CullMode         = CW;

        VertexShader = compile vs_2_0 VertexShaderFunction();
        PixelShader = compile ps_2_0 PixelShaderFunction();
    }

    pass Pass2
    {
        CullMode       = CCW;

        VertexShader = compile vs_2_0 VertexShaderFunction();
        PixelShader = compile ps_2_0 PixelShaderFunction();
    }
}
```

Because you draw 3D models with transparency and you want to be able to see through them, you want to see the inside of the back of the object. Although you normally cull the backward facing triangles, update the cull mode to clockwise so that you will draw the inside triangles. The first effect pass also sets the normal alpha blending states. In the second effect pass, the cull mode is set back to the default counterclockwise so that you can draw the front-facing triangles. If you didn't draw the inside of the objects or you drew the front first, the blending would not look correct if you want the objects to appear to be transparent and to have volume.

The states set by the effect file persists until they are changed either by state objects in the game code or by another effect pass, which defines the state.

Running the sample now displays two spheres with a cylinder between them. Figure 8.20 shows the three objects and how the different transparencies among them enables you to see through the objects to what is behind them.

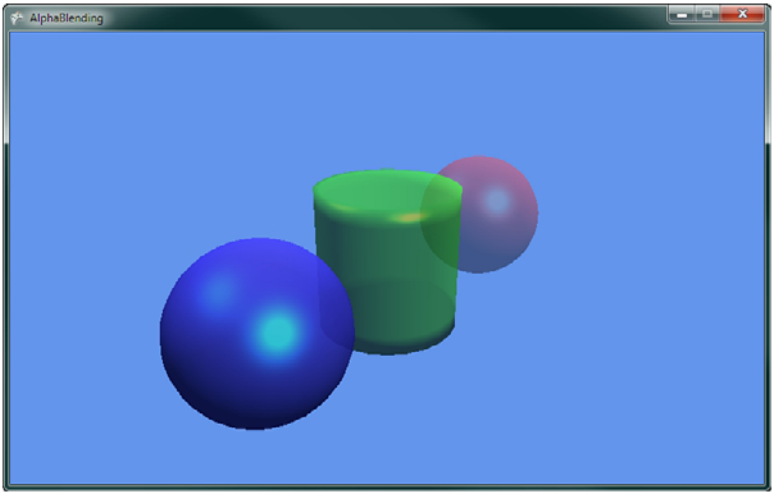


Figure 8.20 Effect states used to draw multiple models with alpha blending

Summary

In this chapter, we covered the basics of creating your own effect files from the different parts of the effect file including the pixel shader, vertex shader, and the techniques and passes. We also covered how to use both vertex colors and textures to color the triangles you draw.

An introduction to different lighting models was covered with numerous samples that built on each other to create useful lighting effects.

Finally, effect states were discussed and how they can be used instead of state objects in your game code to set render states.

This chapter contained many examples in which can be found in the accompanying code files that can be downloaded for this book.

Now that you have a better understanding of what custom effects are and how to use them, you should be able to develop your own effects and be able to better understand other types of shading techniques.

This page intentionally left blank

9

Using the Content Pipeline

In this chapter, you will learn:

- Content processors
- Content importers
- Building assets

New game developers normally don't consider building content. In fact, most don't even realize that content (much like code) should be built to begin with! In the XNA Game Studio runtime, building and subsequently loading content is quite simple.

Even if you didn't realize it, you've been using the content pipeline throughout this book. Every time you load a texture, a model, or any other piece of content, it goes through the content pipeline.You even created a basic content processor in Chapter 6, "Built-In Shader Effects." Many of the intricacies of the content pipeline are discussed throughout this chapter.

Tracing Content Through the Build System

How does content go from an idea in the mind of an artist to something represented in your game? Well, so far, you have added the files to your content projects. Magic happens, and then you load the content in your game! This chapter talks about the magic.

When a file is first added to a content project, its extension is checked. If there is a content importer for the extension, the file's `Build Action` is set to `Compile`. It is now set to go through the content pipeline. During your application's build task, your content also is built.This is done in essentially a two-step process. First, your content is imported, and this is normally done by reading the data in the file on disk, and performing some operation(s) on it.After the data has been imported, it can be optionally processed, which enables you to massage the data if needed.

After the content has been imported and processed, it writes out to a new file that is called `assetname.xnb`, where `assetname` is the name of the asset and `xnb` is the extension (short for XNA Binary).While your game runs, and you call `Content.Load` for that asset, the .xnb binary file opens, reads back in, and the content objects are created. Naturally, this description is high level, and throughout this chapter we delve deeper into these facets.

Why do you need to have a content pipeline? In all actuality, you do not, but without a pipeline means that the importing and processing of the content needs to happen during the runtime of your game! For example, say that you create a 2D game to run on the Xbox 360, you have fifty textures in your game, and each one is 512 by 512 pixels.

You created these textures in an art creation tool (for example, Photoshop or Paint), so they are 32-bit PNG files. Without the content pipeline, you need to open each file and load the data. With so many large textures though, you probably want to use DXT compression on the textures. Because the file you just loaded is a 32-bit PNG file, you will have to do the compression at runtime, and then push the compressed data into the texture. Wait though, you aren't done yet! Your computer processor stores data in little-endian format, but the Xbox 360 CPU expects the data to be big-endian! So you'll also need to byte swap all of the data before you DXT compress it.

Let's say it takes approximately 5ms to read the file from the disk. It then takes another 20ms to swap all the bytes, and another 30ms to DXT compress the texture. Each file is now taking 55ms to process, and with a total of 50 files, you're looking at 2750ms to load the data. That's an almost three-second load time!

Now compare that to a similar scenario using the content pipeline. It still takes 20ms to swap all the bytes and 30ms to DXT compress the texture. However, the 2500ms it takes to do the two operations on 50 files occurs while your game is building, not while it runs. While it runs, it reads only the file off disk. Because the texture is compressed already at build time, reading the file off the disk takes half the time now that the data is smaller! So instead of having an almost three-second load time, your game now has a 125ms load time, which is not even five percent the time it took before. You're doing the same operations, but because you moved the bulk of the work out of the runtime of the game and into the build time, your game is faster.

> **Note**
>
> The time spans used in this example were purely hypothetical and used to illustrate a point. Don't consider them actual measurements.

Now that you have a basic understanding of the flow of the content pipeline (and some of the reasons why you might want to use it), let's dive right in and see some examples.

Content Processors

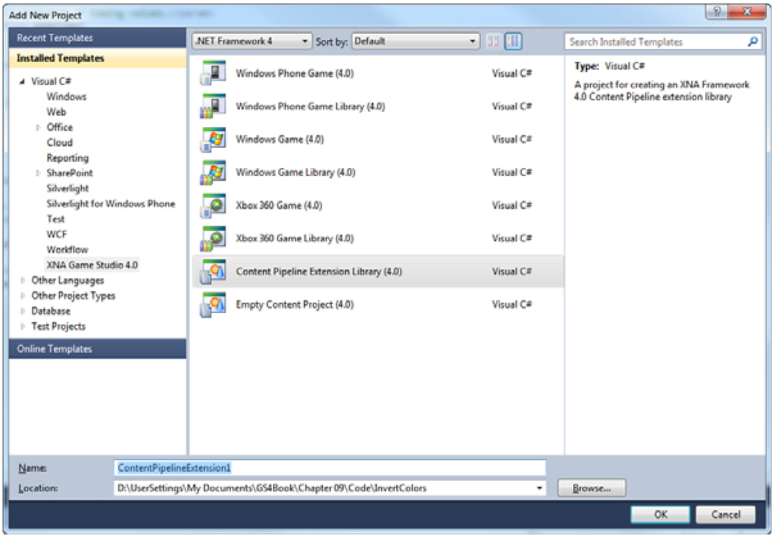
First, let's recreate one of the first examples you did in the book. Create a new `Game` project, and add an image to your content project. The code in the downloadable example uses `cat.jpg`. Then, draw that texture by replacing your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(Content.Load<Texture2D>("cat"),
        GraphicsDevice.Viewport.Bounds, Color.White);
    spriteBatch.End();
}
```

Replace the `cat` asset name in `Content.Load` with whatever asset you used if you used something else. This does nothing more than render your image to fill the entire game window. However, now let's mix things up with a different type of content processor.

> **Note**
>
> You might wonder why the content wasn't stored in a variable and is instead loaded every call to `Draw`. The content manager caches assets for you, so you do not get a new texture every time you draw this.

To add a new project to your solution to hold your content pipeline extension, right-click the solution, choose `Add->New Project`, and choose `Content Pipeline Extension Library` as in Figure 9.1 (you can name it whatever you like).


Add New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Silverlight for Windows Phone
Test
WCF
Workflow
XNA Game Studio 4.0
Other Languages
Other Project Types
Database
Test Projects
Online Templates
Search Installed Templates
Browse...
OK
Cancel

.NET Framework 4
Sort by: Default

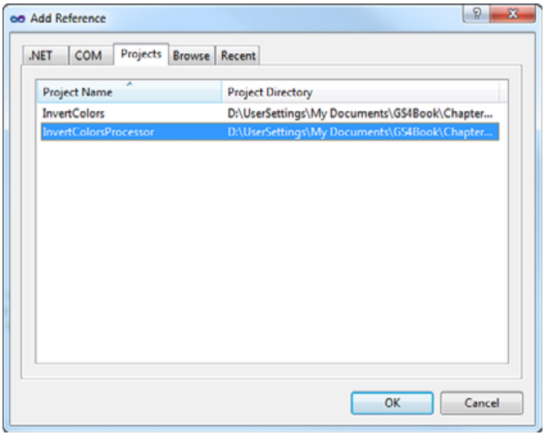
Windows Phone Game (4.0)	Visual C#
Windows Phone Game Library (4.0)	Visual C#
Windows Game (4.0)	Visual C#
Windows Game Library (4.0)	Visual C#
Xbox 360 Game (4.0)	Visual C#
Xbox 360 Game Library (4.0)	Visual C#
Content Pipeline Extension Library (4.0)	Visual C#
Empty Content Project (4.0)	Visual C#

Type: Visual C#
A project for creating an XNA Framework 4.0 Content Pipeline extension library

Name: ContentPipelineExtension1
Location: D:\UserSettings\My Documents\GS4Book\Chapter 09\Code\InvertColors

Figure 9.1 Adding a new content pipeline project

This creates a new content pipeline extension projection in your solution, with a default processor that doesn't actually do much of anything. Because you will update this though, and you want to change how your content is built, you need to add a reference to the new project to your content project. Right-click the `References` in your `Content` project and select `Add Reference`. Then choose the content pipeline extension project you just created from the Projects tab, as in Figure 9.2.



Project Name	Project Directory
InvertColors	D:\UserSettings\My Documents\GS4Book\Chapter...
InvertColorsProcessor	D:\UserSettings\My Documents\GS4Book\Chapter...

Figure 9.2 Adding a reference to your content pipeline project

For this example, you take over the processing phase mentioned earlier. Because you modify the texture, you don't need to import it—you can let the default importer handle that (importers are discussed later in this chapter). To create a content processor, you need to create a class and derive from `ContentProcessor`, so delete the class that was generated for you and replace it with the following:

```
[ContentProcessor(DisplayName = "InvertColorsProcessor")]
public class InvertColorsProcessor : ContentProcessor<Texture2DContent,
➥Texture2DContent>
{
}
```

Notice that the `ContentProcessor` is a generic type with two types required. The first type is the input type of the data that is incoming, and the second is the output type that is returned after processing is complete. Because you modify a texture and return a texture, the input and output types are the same.

Also notice that the types are `Texture2DContent`, not `Texture2D` objects themselves. Although many times the runtime version of an object and the build-time version of an object are the same (as you see later in this chapter), sometimes you might need (or want) to have different properties/data/methods on your build-time type. This is what `Texture2DContent` has, and you see how this gets magically transformed to a `Texture2D` in your game later this chapter.

The last thing to mention before moving on is the first thing your class has, namely the `ContentProcessor` attribute. The `DisplayName` property of this attribute enables you to specify what name is displayed when you see the processor in Visual Studio. The name here implies how you process the texture.

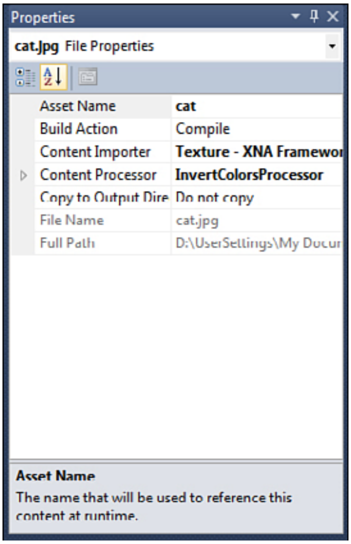
In order to do some actual processing, override the `Process` method in your class. Because it is an abstract method, if you do not do so, you get a compile error. Add the following override to your class:

```
public override Texture2DContent Process(Texture2DContent input,
    ContentProcessorContext context)
{
    // Convert the input to standard Color format, for ease of processing.
    input.ConvertBitmapType(typeof(PixelBitmapContent<Color>));
    foreach (MipmapChain imageFace in input.Faces)
    {
        for (int i = 0; i < imageFace.Count; i++)
        {
            PixelBitmapContent<Color> mip = (PixelBitmapContent<Color>)imageFace[i];
            // Invert the colors
            for (int w = 0; w < mip.Width; w++)
            {
                for (int h = 0; h < mip.Height; h++)
                {
                    Color original = mip.GetPixel(w, h);
                    Color inverted = new Color(255 - original.R,
                        255 - original.G, 255 - original.B);
                    mip.SetPixel(w, h, inverted);
                }
            }
        }
    }
    return input;
}
```

Notice that the `Process` override returns the output type (in this case, `Texture2DContent`) and takes in the input type (again, `Texture2DContent`) as a parameter. It also takes in a `ContentProcessorContext`, which provides helper methods for processing. When the texture is first imported, you don't know what format it is, so first convert it to a known type that you can then modify. The `TextureContent` class (which `Texture2DContent` derives from) luckily includes a method that does this for you. To modify each pixel's color individually, convert to a `PixelBitmapContent` type of `Color`.

After the conversion (if it was even needed), loop through each `MipmapChain` in the texture via the `Faces` property. In the case of a `Texture2DContent`, it is only a single `Face`, and a cube texture has six. You then can enumerate through each mip level in the face. After you have the current mip level, get each pixel's color by using the `GetPixel` method, create a new color that is an inversion of the original color, and then use the `SetPixel` method to update with the new inverted color. At the end, return the `Texture2DContent` you are modifying and you're done. You now have a content processor that will invert all of the colors in a texture.

Notice that when running the example, nothing at all has changed. The colors certainly aren't inverted; it's the same image as it was last time! That's because you never changed the actual processor your application uses. Select the image you added to the content project and update its `Content Processor` to the `InvertColorsProcessor`, as in Figure 9.3. The name shown here is whatever you used for the `DisplayName` in the attribute before your class previously.



Asset Name	**cat**
Build Action	Compile
Content Importer	**Texture - XNA Framewo**
Content Processor	**InvertColorsProcessor**
Copy to Output Dire	Do not copy
File Name	cat.jpg
Full Path	D:\UserSettings\My Docu

Asset Name
The name that will be used to reference this content at runtime.

Figure 9.3 Choosing your processor

Now when you run the application, notice that your original image is shown with the colors inverted because the processor you wrote is used, as in Figure 9.4.

Now, this is all well and good, but it isn't customizable. The image simply has its colors inverted. A way to customize the processors would be useful, and luckily, you can do just that. Add a couple properties to control how the colors are inverted (allow them to form blocks):

```
[DefaultValue(false)]
[DisplayName("BlockyInversion")]
[Description("Should the inversion be done in blocks?")]
public bool ShouldCauseBlocks
{
    get;
    set;
}
[DefaultValue(20)]
[DisplayName("BlockSize")]
```

```
[Description("The size in pixels of the blocks if a blocky inversion is done")]
public int BlockSize
{
    get;
    set;
}
```

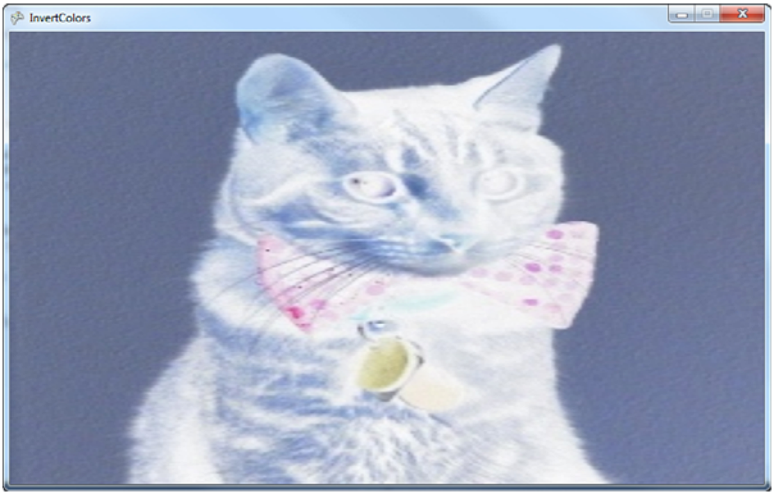


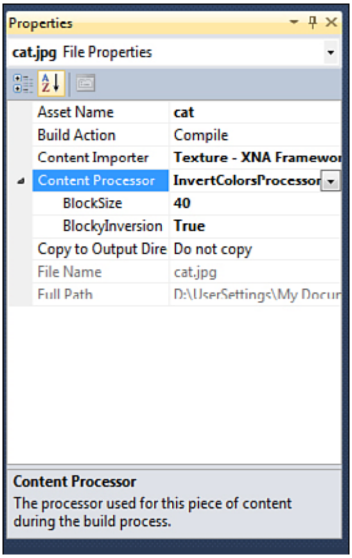
Figure 9.4 The image with the colors inverted

The properties themselves are simple enough—a `bool` to determine whether you should use the blocky code (that you haven't written yet), and an `int` to specify the size of the blocks. Notice that the attributes on the properties enable you to control how the properties are displayed in Visual Studio. If you compile your solution now, and then look at the properties of your image in the content project, you now see two extra properties available, as in Figure 9.5.

> **Note**
>
> If you get compile errors on the attributes, add a using `System.ComponentModel` clause to the code file.

Notice how the property name in the Visual Studio window is the name specified in the `DisplayName` attribute, and not the real property name. Set the `BlockyInversion` property to true. To update the processor to respond to the properties, add the following code before the `SetPixel` call in your `Process` method:

```
if (ShouldCauseBlocks)
{
    if ((h % BlockSize > (BlockSize / 2))
        || (w % BlockSize > (BlockSize / 2)))
        {
            inverted = original;
        }
    }
```



Asset Name	cat
Build Action	Compile
Content Importer	Texture - XNA Framewo
Content Processor	InvertColorsProcessor
BlockSize	40
BlockyInversion	True
Copy to Output Dire	Do not copy
File Name	cat.jpg
Full Path	D:\UserSettings\My Docur

Content Processor
The processor used for this piece of content during the build process.

Figure 9.5 Content processor properties

Running the application now causes your image to be partially inverted, and partially not, forming a blocky type pattern, as in Figure 9.6. Remember that all of the inversion is done at compile time. The texture that is loaded into memory is already modified, so none of this happens at runtime.

Debugging Content Pipeline Extensions

Because the content pipeline is executed during compilation time, you can't just "run the application" to debug the code you're writing in your content pipeline extension. In reality, the application that is running the code is Visual Studio (or MSBuild, depending on how you're building), which means that you need to do something else for debugging.

One option is to use a second version of Visual Studio (or whichever debugger you use), and use it's Attach to Process command to attach to the original Visual Studio instance. This enables you to put break points in your content pipeline extension code and debug. However, this makes Visual Studio run remarkably slow, so it isn't recommended.

An easier solution is to force the compilation of your content to give you the opportunity to debug, which you can do using the `Debugger.Launch` method in the `System.Diagnostics` namespace. When the line of your code is executed, it forces the system to attempt to launch the debugger, and you see a dialog much like Figure 9.7. Using this method, you can then debug your content pipeline extension.

Note that if you select No in this dialog, it kills your Visual Studio session.

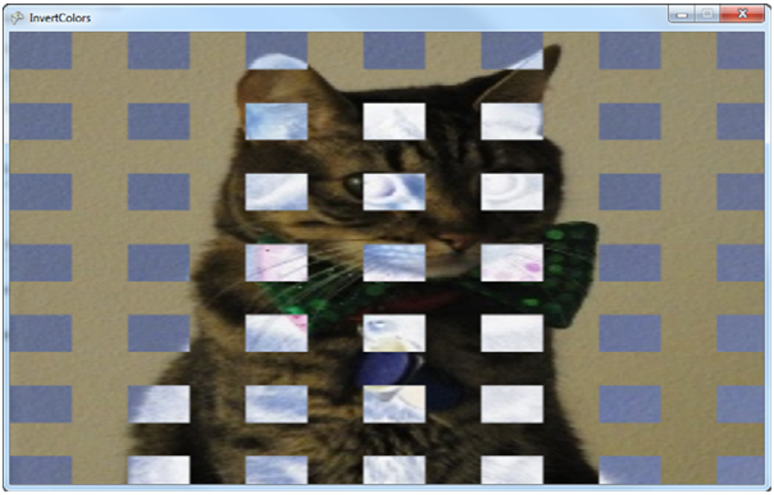


Figure 9.6 A blocky image inversion

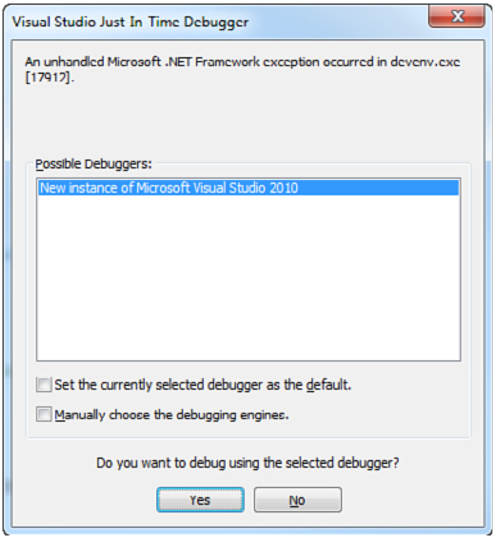
Content Importers

Using a file type that the content pipeline already imports is one thing; there are times when you have a completely new type of file to import (for example, a level in your game). So let's create an example that shows off a concept such as that. First, create a new `Game` project. You're actually going to create quite a few projects here, so add a new `Content Pipeline Extension` project to the solution as well that will hold your importer. Lastly, add another project to your solution, but this time a `Game Library` project. This is used to hold the data types that both your game and the content pipeline extension will need.

Set up your project references correctly as well. Add a reference to the game library project you just created to both the content pipeline extension project as well as the game project. Add a reference to the content pipeline extension project to your game's content project as in the previous example.

Now, let's create the shared data type that will represent your game's level. In your game library project, replace the class and namespace that it auto-generated with the following:

```
namespace ContentImporter
{
    public class Level
    {
        public List<Vector3> PositionList;
        public List<Vector3> RotationList;
    }
}
```



An unhandled Microsoft .NET Framework exception occurred in devenv.exe [17912].

Possible Debuggers:

New instance of Microsoft Visual Studio 2010

☐ Set the currently selected debugger as the default.

☐ Manually choose the debugging engines.

Do you want to debug using the selected debugger?

Figure 9.7 Debugging a content pipeline extension

You don't need anything fancy for this example. Simply hold a couple lists of `Vector3`, one of which holds the position of each model in your world, and the other holds the rotation portion. Before the content pipeline extension portion, let's implement the game portion. Add the following variables to your game project:

```
Level mylevel;
Model model;
Matrix view;
Matrix proj;
```

Later in the chapter, you learn how to include things such as `Model` objects in your runtime classes, but for now add a model to your content project. The downloadable example uses the `depthmodel.fbx` model. In your `LoadContent` method, create the following objects:

```
view = Matrix.CreateLookAt(new Vector3(0, 10, 150), Vector3.Zero, Vector3.Up);
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio,
    1.0f, 1000.0f);
model = Content.Load<Model>("depthmodel");
mylevel = Content.Load<Level>("mylevel");
```

Finally, replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
        }
    }
    for(int i = 0; i < mylevel.PositionList.Count; i++)
    {
        model.Draw(Matrix.CreateTranslation(
            mylevel.PositionList[i]) * Matrix.CreateFromYawPitchRoll(
            mylevel.RotationList[i].X, mylevel.RotationList[i].Y,
            mylevel.RotationList[i].Z), view, proj);
    }
}
```

Here, you use the data from your level to render the model wherever it says after enabling the default lighting model. With that out of the way though, you are almost ready to create your content pipeline extension. First, add a new file to your content project called `mylevel.mylevel`, and include the following data in it:

```
10 10 10 0.3 0.2 0.1
20 20 20 0.1 0.2 0.3
30 30 30 0.2 0.3 0.1
1.5 2.3 1.7 0.4 0.5 0.6
40.2 70.1 -10.2 0.6 0.5 0.4
-10 10 -10 0.7 0.8 0.9
20 -20 20 0.9 0.8 0.7
-30 30 -30 0.8 0.9 0.7
```

This is the data you are going to import that represents your level. Each line is six different numeric values. The first three represents the position in the world, and the next three represents the yaw, pitch, and roll angles of rotation for the model. Now, go to your content pipeline extension project and remove the default `ContentProcessor` class it auto-generated, and replace it with the following:

```
[ContentImporter(".mylevel", DisplayName="My Level Importer")]
public class MyContentImporter : ContentImporter<ContentImporter.Level>
{
}
```

Much like your processor, use an attribute to describe your importer. The first parameter is the extension of the file the class will import; in this case, we picked `.mylevel`. You also need to derive from the `ContentImporter` class using the type you will fill with the imported data. This object has a single abstract method (again much like the processor) called Import. You can use this implementation of the method:

```
public override ContentImporter.Level Import(string filename,
    ContentImporterContext context)
```

```
{
    ContentImporter.Level level = new ContentImporter.Level();
    level.PositionList = new List<Vector3>();
    level.RotationList = new List<Vector3>();
    using (StreamReader reader = new StreamReader(File.OpenRead(filename)))
    {
        while (!reader.EndOfStream)
        {
            string current = reader.ReadLine();
            string[] floats = current.Split(‘ ‘);
            level.PositionList.Add(new Vector3(float.Parse(floats[0]),
                float.Parse(floats[1]), float.Parse(floats[2])));
            level.RotationList.Add(new Vector3(float.Parse(floats[3]),
                float.Parse(floats[4]), float.Parse(floats[5])));
        }
    }
    return level;
}
```

> **Note**
>
> If you have compilation errors with the `StreamReader` class, add a using clause at the top of the code file for `System.IO`.

Notice that the method is quite simple. You create your `level` object that you'll return after importing the data. You create the two lists that will store the data. You then use the standard runtimes `StreamReader` class to read through the file (which is passed in as a parameter to the import method). You split each line of the file into the individual numbers they contain and drop those into your two lists, and you are done.

Now, this code doesn't check errors, so if something goes wrong, it won't work. We discuss ways to handle this in a few moments. Build the content pipeline extension project, go back to your content project, and notice that your `mylevel.mylevel` file uses the My Level Importer importer. If it is not, select the content importer, and make sure that the `Build Action` is set to `Compile`. If you run your application now, you see your level in action, as in Figure 9.8.

So as you see, this works, but who would want to build a level that way? You still have to create a model in your game, and you use only a single model in the level. It would be better to make your importer work, and import models and textures and everything all in one go to make a much better level.

Combining It All and Building Assets

Although the concepts are slightly more in depth than what you've seen up until now, that's exactly what you're about to do. Create a new solution and perform the same steps you did in the previous section, except skip the extra game library project. So, create a new game project, a new content pipeline extension project, and add a reference to the extension project to your game's content project. Later in this section, you see why you don't need to create the extra game library project for this example.

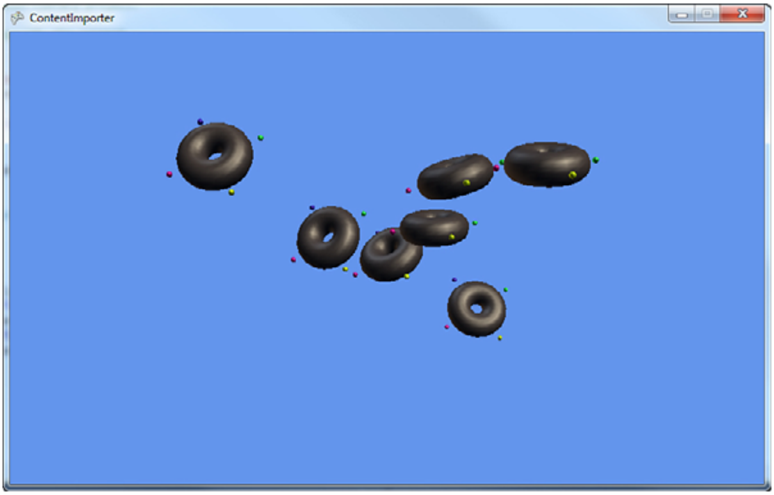


Figure 9.8 Loading a world from a content importer

Again, let's start with what your game project will do. Add the following classes that will represent your level to your game project:

```
public class MyLevelObject
{
    public Model Model;
    public Vector3 Position;
    public Vector3 Rotation;
}
public class Level
{
    public List<MyLevelObject> LevelObjects;
    public Texture2D Background;
}
```

Although this still isn't a realistic object model that you would load a level into, it is much closer than the previous example. The `Level` object itself stores a texture that is used as the background of your level, and a list of objects that describe the entities in your level. In this case, each entity is a `MyLevelObject` that includes the `Model` that is used, along with its `Position` and `Rotation` like the previous example. Let's write the game portion now, so add the following variables to your `Game` class.

```
Level mylevel;
Matrix view;
Matrix proj;
```

Notice that you don't need the `Model` variable this time because it is stored inside the `Level` object. You also don't have anything declared for your background texture, because this is also a part of the `Level` object. Create the following objects in your `LoadContent` overload:

```
view = Matrix.CreateLookAt(new Vector3(0, 10, 150), Vector3.Zero, Vector3.Up);
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio,
    1.0f, 1000.0f);
mylevel = Content.Load<Level>("mylevel");
```

At the end of this section, the last line loads the whole level, including all of the models, textures, and data. Before learning how to make this work, let's finish the game portion by replacing the `Draw` method with the following one:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(mylevel.Background, GraphicsDevice.Viewport.Bounds,Color.White);
    spriteBatch.End();
    foreach (MyLevelObject myObject in mylevel.LevelObjects)
    {
        foreach (ModelMesh mesh in myObject.Model.Meshes)
        {
            foreach (BasicEffect effect in mesh.Effects)
            {
                effect.EnableDefaultLighting();
            }
        }
        myObject.Model.Draw(Matrix.CreateTranslation(
                myObject.Position) * Matrix.CreateFromYawPitchRoll(
                myObject.Rotation.X, myObject.Rotation.Y,
                myObject.Rotation.Z), view, proj);
    }
}
```

This is similar to the previous method, except you first use the `Background` property of your level to render a texture covering the entire game. You then make sure that each object in your level has the lighting enabled, and then render each object based on the data found in your level.

From here, things get quite a bit more esoteric. First, notice that the `Level` object is declared in your Game project this time, rather than a shared project that both the content pipeline extension and the game project has access to. This is because there are certain types where you need (or want) to have a different type of object used at build time than you want to use at run time.

You experienced this earlier in the chapter. If you recall, your first content processor was eventually turned into a `Texture2D` in your game, but it started out as a `TextureContent` in the content pipeline. `TextureContent` is the build-time version of the `Texture` runtime class. This is because there are things on the build-time version of the class that make no sense in the runtime version (for example, the conversion helper function you used earlier). Knowing this, because your objects use these types, you also need build-time versions, so declare your versions now in your content pipeline extension project:

```
public class MyLevelObject
{
    public ModelContent Model;
    public Vector3 Position;
    public Vector3 Rotation;
    public string ModelFileName;
}
public class Level
{
    public List<MyLevelObject> LevelObjects;
    public TextureContent Background;
    public string BackgroundFileName;
}
```

Notice that this is almost an exact duplicate of your runtime class, except the graphics objects have been replaced with the build-time versions of themselves (in this case, `ModelContent` and `TextureContent`). Each object also includes a new property that is a string representing the `filename` needed for each of the graphics objects. You see why this is needed in a few moments.

Before you create your importer and processor for these objects, let's create the actual data you'll use. Create a new file in your content project called `mylevel.mylevel` and include the following data:

```
glacier.jpg
depthmodel.fbx
10 10 10 0.3 0.2 0.1
environmentmapmodel.fbx
20 20 20 0.1 0.2 0.3
depthmodel.fbx
-50 -40 30 0.2 0.3 0.1
box.fbx
1.5 2.3 1.7 0.4 0.5 0.6
environmentmapmodel.fbx
40.2 70.1 -10.2 0.6 0.5 0.4
depthmodel.fbx
-10 10 -10 0.7 0.8 0.9
```

```
depthmodel.fbx
20 -20 20 0.9 0.8 0.7
environmentmapmodel.fbx
-30 30 -30 0.8 0.9 0.7
```

The first line is the filename of the texture that is used as the background for the level. Each pair of lines represents first the model filename and then the data for that associated model. You can use any valid model and texture filename here. Note, however, that these files are expected to be in the same folder as the `mylevel.mylevel` file, so if they aren't, copy them there now (they are already in the same folder in the example provided with the downloadable examples).

Next, add the `ContentImporter` class to your extension project:

```
[ContentImporter(".mylevel", DisplayName = "My Level Importer",
    DefaultProcessor = "MyLevelProcessor")]
public class MyContentImporter : ContentImporter<Level>
{
    public override Level Import(string filename,
        ContentImporterContext context)
    {
        Level level = new Level();
        level.LevelObjects = new List<MyLevelObject>();
        using (StreamReader reader = new StreamReader(File.OpenRead(filename)))
        {
            // The first line is the background texture
            level.BackgroundFileName = GetAbsolutePath(filename,
➥reader.ReadLine());
            while (!reader.EndOfStream)
            {
                // After the first line, each line is first the model,
                // and then the model's data
                MyLevelObject info = new MyLevelObject();
                info.ModelFileName = GetAbsolutePath(filename, reader.ReadLine());
                // Followed by the position/rotation data
                string[] floats = reader.ReadLine().Split(‘ ’);
                info.Position = new Vector3(float.Parse(floats[0]),
                    float.Parse(floats[1]), float.Parse(floats[2]));
                info.Rotation = new Vector3(float.Parse(floats[3]),
                    float.Parse(floats[4]), float.Parse(floats[5]));
                level.LevelObjects.Add(info);
            }
        }
        return level;
    }
    private string GetAbsolutePath(string input, string file)
    {
        return Path.Combine(Path.GetDirectoryName(input), file);
    }
}
```

First, notice that the attribute for your importer class has a new property added, the `DefaultProcessor`. This is used to specify the name of the default processor to be used by things that the importer will import. This is the name of the processor you'll write in a few moments.

There is also a helper method included that converts the filename you used in the actual data you created to the absolute location on disk by using the folder the data file was found in. This is why you need to make sure the textures and models your level needs are in the same folder as your level file (or change this code so it knows where to find them).

Aside from that, this is almost the same as your other importer except that you are not yet creating the textures and models your level needs, but instead simply storing the full filenames in the extra property. This is because you do that portion in the processor. Because there's nothing else new here, let's add the processor to this same code file now:

```
[ContentProcessor(DisplayName="MyLevelProcessor")]
public class MyContentProcessor : ContentProcessor<Level, Level>
{
    public override Level Process(Level input, ContentProcessorContext context)
    {
        Level level = new Level();
        level.LevelObjects = new List<MyLevelObject>();
        // Process the input, we'll need to create the texture that is specified
        ExternalReference<TextureContent> backgroundReference =
            new ExternalReference<TextureContent>(input.BackgroundFileName);
        // Now build and store it
        level.Background = context.BuildAndLoadAsset<TextureContent,
            TextureContent>(backgroundReference, "TextureProcessor");
        // Now load each of the models
        foreach (MyLevelObject obj in input.LevelObjects)
        {
            MyLevelObject newobj = new MyLevelObject();
            newobj.Position = obj.Position;
            newobj.Rotation = obj.Rotation;
            // Do the same thing we did for textures for model here
            ExternalReference<NodeContent> modelReference =
                new ExternalReference<NodeContent>(obj.ModelFileName);
            // Now build and store it
            newobj.Model = context.BuildAndLoadAsset<NodeContent,
                ModelContent>(modelReference, "ModelProcessor");
            level.LevelObjects.Add(newobj);
        }
        return level;
    }
}
```

This is different from anything you wrote previously, and there is quite a bit of stuff going on here. First, create the `Level` you'll return and the `LevelObjects` that go with it. Next, load the file you know about for the background texture into an actual `TextureContent` object. To do this, create an `ExternalReference` object of the type you want to load—in this case, `TextureContent`. You can then use the `BuildAndLoadAsset` helper method, which builds the asset specified in the external reference, and then load its into the appropriate processed type, which you then can store in your new `Level` object as the `Background` texture. By using the `BuildAndLoadAsset` helper method, this also adds the external file as a dependency of your `Level` file, which enables the content to be recreated if anything is modified.

After you load the `TextureContent`, you are ready to move on to the object list. Copy the `Position` and `Rotation` over, and then perform a similar operation to the `Model` as you did to the `Background` earlier. The major difference here is that the model is imported as a `NodeContent` object, and then processed into the `ModelContent` you need rather than being imported and processed as the same type. Then, add your new data into the list before finally returning the level.

You might wonder, "Ok, I have the importer and the processor written, now I can run my application and see the work!" If you do that, you get an exception when you try to load your level. It complains about not being able to find the `Level` type. This is the reason your first importer had the extra game library project that stored the type, so both build time and runtime had access to it. However, you can customize how your content assets are written out to disk by using a `ContentTypeWriter`. Add the following using clause at the top of your content pipeline project:

```
using Microsoft.Xna.Framework.Content.Pipeline.Serialization.Compiler;
```

Then, declare a new class that will be the writer for your content:

```
[ContentTypeWriter]
public class LevelContentWriter : ContentTypeWriter<Level>
{
}
```

This attribute tells the system that this class should be checked before writing content, and if the content you write is the appropriate type (the type specified in the generic of the `ContentTypeWriter` class), then to use this class rather than the default writer. Because this is the writer for the `Level` class, anytime a `Level` asset is written, this class will be called instead. There are three methods that need to be overwritten for this class to work. Add the following:

```
public override string GetRuntimeReader(TargetPlatform targetPlatform)
{
    return "ContentImporter.LevelContentReader, ContentImporter";
}
public override string GetRuntimeType(TargetPlatform targetPlatform)
{
    return "ContentImporter.Level, ContentImporter";
}
protected override void Write(ContentWriter output, Level value)
{
    // Write the background first
    output.WriteObject<TextureContent>(value.Background);
    // Next write the count, so we know how many models there are
    output.Write(value.LevelObjects.Count);
    // Finally, write all the level objects
    foreach (MyLevelObject obj in value.LevelObjects)
    {
        output.WriteObject<ModelContent>(obj.Model);
        output.Write(obj.Position);
        output.Write(obj.Rotation);
    }
}
```

Because you have a customized writer now, you need a customized reader. The first method enables you to specify what to use as the reader for this type of content. It expects you to return this as the assembly qualified name of the reader class. In this case, the game project is called `ContentImporter`, and the namespace the reader class lives in (after you write in shortly) is also `ContentImporter`, so this is the assembly qualified name of this type. If you aren't sure what the name of the type is, you can verify it by using the `typeof(Type).AssemblyQualifiedName` helper property on the appropriate type.

The next method is also the assembly qualified name of a type, but this time it is the runtime version of this content type. Because you can use the `GetRuntimeType` method to specify the type that you will load from the data, you don't need to have the shared game library project.

Finally, you have to actually write the data, which is what the `Write` overload does. You simply write the `Background` first, then the number of objects in your list, followed by each object itself. Now, you're ready to see the fruits of all this labor. All you need now is the reader half of this and you're done. Add the following class back in your game project:

```
public class LevelContentReader : ContentTypeReader<Level>
{
    protected override Level Read(ContentReader input, Level existingInstance)
    {
            Level level = new Level();
            level.Background = input.ReadObject<Texture2D>();
            int count = input.ReadInt32();
            level.LevelObjects = new List<MyLevelObject>(count);
            for (int i = 0; i < count; i++)
            {
                MyLevelObject obj = new MyLevelObject();
                obj.Model = input.ReadObject<Model>();
                obj.Position = input.ReadVector3();
                obj.Rotation = input.ReadVector3();
                level.LevelObjects.Add(obj);
            }
            return level;
        }
    }
```

As you see here, you create the new Level object, and then reverse what you did in the writer. You read the `Background` first, followed by the number of items in the list, before finally reading each list object and adding them to your new list. If you run the application now, it builds everything up into your `Level` object, your scene renders on the screen much like Figure 9.9.

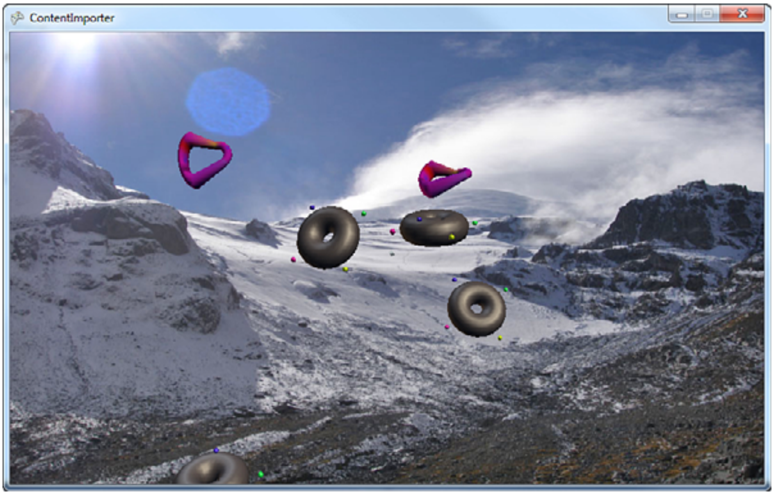


Figure 9.9 Your full level rendered

Combining What You Learned So Far

What if you want to combine various importers and processors though? What if you wanted to use the color inversion processor on your background texture for your level? Luckily, you can do that, too! Add the final version of the color inversion processor to your content pipeline extension project:

```
[ContentProcessor(DisplayName = "InvertColorsProcessor")]
public class InvertColorsProcessor : ContentProcessor<TextureContent,
➥TextureContent>
{
    public override TextureContent Process(TextureContent input,
        ContentProcessorContext context)
    {
        // Convert the input to standard Color format, for ease of processing.
        input.ConvertBitmapType(typeof(PixelBitmapContent<Color>));
        foreach (MipmapChain imageFace in input.Faces)
        {
            for (int i = 0; i < imageFace.Count; i++)

                PixelBitmapContent<Color> mip = (PixelBitmapContent<Color>)
➥imageFace[i];
                // Invert the colors
                for (int w = 0; w < mip.Width; w++)
                {
                    for (int h = 0; h < mip.Height; h++)
                    {
                        Color original = mip.GetPixel(w, h);
                        Color inverted = new Color(255 - original.R,
                            255 - original.G, 255 - original.B);
                        if (ShouldCauseBlocks)
                        {
                            if ((h % BlockSize > (BlockSize / 2))
                                || (w % BlockSize > (BlockSize / 2)))
                            {
                                inverted = original;
                            }
                        }
                        mip.SetPixel(w, h, inverted);
                    }
                }
            }
        }
        return input;
    }
    [DefaultValue(false)]
    [DisplayName("BlockyInversion")]
    [Description("Should the inversion be done in blocks?")]
    public bool ShouldCauseBlocks
    {
        get;
        set;
    }
    [DefaultValue(20)]
    [DisplayName("BlockSize")]
    [Description("The size in pixels of the blocks if a blocky inversion is done")]
    public int BlockSize
    {
        get;
        set;
    }
}
```

Now you will want to include the same properties on your `Level` processor, but also include another one to specify whether the level background should be inverted or not. Add the following properties to the `MyContentProcessor` class:

```
[DefaultValue(false)]
[DisplayName("ShouldInvertColors")]
[Description("Should the background texture be inverted?")]
public bool ShouldTexturesBeInverted
{
    get;
    set;
}
[DefaultValue(false)]
[DisplayName("BlockyInversion")]
[Description("Should the inversion be done in blocks?")]
public bool ShouldCauseBlocks
{
    get;
    set;
}
[DefaultValue(20)]
[DisplayName("BlockSize")]
[Description("The size in pixels of the blocks if a blocky inversion is done")]
public int BlockSize
{
    get;
    set;
}
```

To invert the background texture (if it was specified as such), add the following code to the beginning of the `Process` overload of your `MyContentProcessor` class:

```
string textureProcessor = ShouldTexturesBeInverted ?
    "InvertColorsProcessor" : "TextureProcessor";
OpaqueDataDictionary opaqueData = new OpaqueDataDictionary();
if (ShouldTexturesBeInverted)
{
    opaqueData.Add("ShouldCauseBlocks", ShouldCauseBlocks);
    opaqueData.Add("BlockSize", BlockSize);
}
```

First, choose which processor should be used based on whether or not the user has asked to have the textures inverted. Then, create a new `OpaqueDataDictionary` that is used to store extra data for the processor if it is needed. Any data in the dictionary is used to populate the properties of the processor before it is executed. So if the user has asked to invert the colors, you pass in the extra parameters specifying whether it should be blocky or not, and by how much.

To update your building command to use these new properties, replace the first `BuildAndLoadAsset` method with the following:

```
level.Background = context.BuildAndLoadAsset<TextureContent,
    TextureContent>(backgroundReference, textureProcessor, opaqueData, null);
```

This passes in the correct processor and opaque data depending on which options were specified. Compile this now, and then change the properties of your level to allow color inversion of the background. Compile and run to see that the background image is now inverted!

The Content Manager

The `ContentManager` class is part of the runtime portion of the Content Pipeline. It is the class that you use to load the content. It also caches the content it loads, so subsequent requests for the same piece of content simply returns it again, so you don't need to worry about it loading multiple copies of the same thing. This also means that when you call `Unload` on a `ContentManager`, it destroys all the assets it has loaded in one swoop.

More importantly, you can create as many `ContentManager` classes as you want, and use them to control how your game manages asset lifetime. For example, to have one `ContentManager` that holds all of the user interface elements that are needed for menus, startup, and so on, use a separate `ContentManager` to hold the assets for your content that is part of the game and levels. Then, when a level is over, you can `Unload` the `ContentManager` and free up all the resources you were using.

Although all of the examples in this book use a single `ContentManager` class (and the default game templates do as well), don't assume that it means you should use only one. More complex games should use several of these and unload them at appropriate times.

Summary

The content pipeline is a powerful tool in your arsenal of creating great games. Moving your asset loading from runtime to build time can give you huge gains in performance, and makes your game code clean and easy to read. This, coupled with the fact that the content pipeline is so extensible, enables you great freedom in how you use the pipeline for your content.

In the next chapter, you learn everything there is to know about Avatars!

10

Having Fun with Avatars

In November of 2008, the Xbox 360 added a feature to enable users to add a stylized cartoonish character representation of themselves to their Xbox Live account called an Avatar. The avatars can be extensively customized by a user. There are numerous body sizes, shapes, and features to choose from. In addition, users have the capability to choose the clothing or costume for their avatars. There is a standard set of clothing items available to everyone in addition to items that are available only by unlocking them in games or purchasing them from the avatar marketplace.

In this chapter you will learn about:

- How to use animation and render an avatar
- Lighting avatars
- Playing multiple animations at once
- Blending between animations
- How to place objects onto avatars
- Using avatars with render targets
- How to create and use custom animations

XNA Game Studio 4.0 provides easy-to-use APIs that enable you to animate and render the avatar for the players in your game or to generate random avatars for users within your game. Avatars provide an easy way to add characters to your game without requiring an artist to create numerous character models. They also provide a level of user customization to your game that users find enjoyable.

Avatars are a popular feature used in many of the top Xbox Live Indie games.

Introduction to Avatars

The avatar APIs in XNA Game Studio 4.0 provide a high-level abstraction that provides the capability to animate and render avatars with a simple API. The process of loading and managing the models and textures used to draw the avatars is hidden from the developer. Figure 10.1 shows what a avatar looks like.

Figure 10.1 Example of a player's avatar

The avatar functionality is included in the Microsoft.Xna.Framework.Avatars assembly, which is available to the Xbox 360 and Windows platforms. Although the assembly is available on Windows, it is not possible to draw avatars on Windows. The Windows APIs are provided to minimize the amount of platform-specific defines needed in your code.

Accessing Avatar Information Using AvatarDescription

An avatar has many customizable components from gender, height, size, clothing, and accessories. Each user has his or her own `AvatarDescription` that stores all of the customizable choices the user has made when building the avatar. The `AvatarDescription` contains an internal opaque data buffer that stores all of these choices. The `AvatarDescription` is used to tell the `AvatarRender` which model assets and textures should be used for a specific avatar.

Accessing a Gamer's AvatarDescription

The `AvatarDescription` class provides an asynchronous method `BeginGetFromGamer` to access the `AvatarDescription` for any type that inherits from the `Gamer` base type. The most common scenario in your game is to load the `AvatarDescription` for one of the `SignedInGamer` players in your game.

Let's see how to access an `AvatarDescription` from a `SignedInGamer`. First, add a member variable to your games class to store the `AvatarDescription`.

```
AvatarDescription avatarDescription;
```

Next, in the games constructor, add the following two lines of code.

```
Components.Add(new GamerServicesComponent(this));

SignedInGamer.SignedIn += new EventHandler<SignedInEventArgs>(SignedInGamer_
➥SignedIn);
```

The first line adds the `GamerServiceComponent` to the game's `Components` list. This ensures that the `GamerServicesDispatcher` is regularly updated as the game runs. The `GamerServicesComponent` is discussed in more detail in Chapter 15, "Gamer Services."

The second line creates a new event handler that is fired anytime a player signs into the game. Even if a player is signed into Xbox when your game launches, the `SignedInGamers` collection will not populate for a number of frames. You can poll the collection yourself, but signing up for this event ensures you are notified anytime a player signs into the console.

The next bit of code to add is the new `SignedInGamer_SignedIn` method that is called when the `SignedIn` event fires. Add the following method to your game class:

```
void SignedInGamer_SignedIn(object sender, SignedInEventArgs e)
{
    // Only handle player one sign in
    if (e.Gamer.PlayerIndex == PlayerIndex.One)
    {
        AvatarDescription.BeginGetFromGamer(e.Gamer, LoadGamerAvatar, null);
    }
}
```

For this example, you want to request only the first player's `AvatarDescription`, so check the `SignedInEventArgs PlayerIndex` property to see whether this event was fired for the first player. If you have the correct player, you then kick off the `BeginGetFromGamer` asynchronous method. `BeginGetFromGamer` is passed the instance of the `Gamer` object and a `AsyncCallback` method that you will implement.

To implement the `LoadGamerAvatar AsyncCallback` method, add the following method to your game's class.

```
void LoadGamerAvatar(IAsyncResult result)
{
    // Get the AvatarDescription for the gamer
    avatarDescription = AvatarDescription.EndGetFromGamer(result);

    // Is the AvatarDescription valid
    if (avatarDescription.IsValid)
        // Use the AvatarDescription and load AvatarRenderer
}
```

In the callback method, first call `EndGetFromGamer` method, which returns `AvatarDescription`. After the `AvatarDescription` has been returned, it is possible that the user did not have a valid avatar set for his or her profile. In that case, the `IsValid` property returns `false`. On `Windows` the returned `AvatarDescription` always returns false for `IsValid`.

Creating a Random AvatarDescription

Along with the players' avatars, you might also want to display more avatars to represent other nonpayer characters in your game. In that case, you can create random avatars.

To create a random `AvatarDescription`, the static `CreateRandom` method is provided.

```
AvatarDescription description = AvatarDescription.CreateRandom();
```

This creates either a random female or male avatar that your game can then use. If you want to specify a gender for the avatar, you can use the `CreateRandom` overload, which takes an `AvatarBodyType` enumeration argument. The `AvatarBodyType` enumeration has two values of `Female` and `Male`.

```
AvatarDescription description = AvatarDescription.CreateRandom(AvatarBodyType.
➥Female);
```

The `AvatarDescription.BodyType` property can be used to determine the gender of an existing `AvatarDescription`. It returns the `AvatarBodyType` for the already created `AvatarDescription`.

The `AvatarDescription.Height` property returns a `float` value representing the avatar's height in meters. This is the size of the avatar skeleton from the bottom of the feet to the top of the head. This does not take into account any type of hat or other clothing item that might be on the head.

Constructing an AvatarDescription from a Byte Array

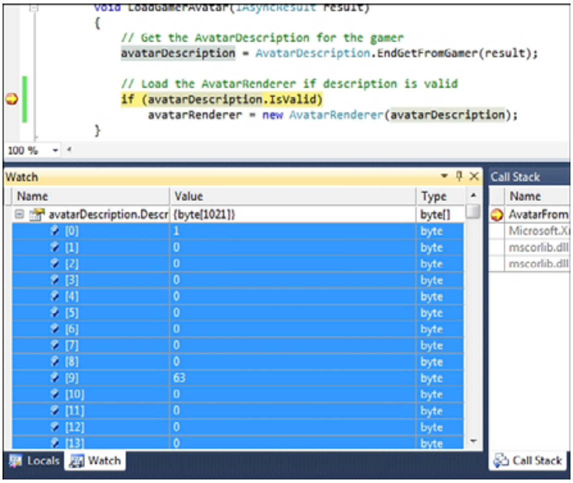
The `AvatarDescription` class provides a third way to create a new instance of the type. It provides a constructor that takes the opaque description as a `byte` array. This constructor can be useful in a couple of scenarios. The first is to enable the player to select random characters in your game and then save his or her selection so the same avatar can be used across play sessions.

To obtain the byte array from the existing `AvatarDescription`, use the following `Description` property:

```
byte[] opaqueBlob = avatarDescription.Description;
AvatarDescription copyAvatarDescription = new AvatarDescription(opaqueBlob);
```

The second scenario where the `byte` array constructor is useful is for customizing the nonplayer character avatars in your game. For example, you have a shopkeeper in your game that you want to customize to have a beard and top hat. This can be accomplished by first customizing the avatar in the dashboard avatar editor. Then, load the `AvatarDescription` for that player. Next, use the debugger to print out the `Description` property to the debugger. Figure 10.2 shows the `AvatarDescription` in the debug window.

You can then copy, paste, and format the values into a `byte` array in your source code. This hard-coded array can then be used to create a new instance of the `AvatarDescription`.



```
void LoadGamerAvatar(IAsyncResult result)
{
    // Get the AvatarDescription for the gamer
    avatarDescription = AvatarDescription.EndGetFromGamer(result);

    // Load the AvatarRenderer if description is valid
    if (avatarDescription.IsValid)
        avatarRenderer = new AvatarRenderer(avatarDescription);
}
```

Name	Value	Type
avatarDescription.Descr	{byte[1021]}	byte[]
[0]	1	byte
[1]	0	byte
[2]	0	byte
[3]	0	byte
[4]	0	byte
[5]	0	byte
[6]	0	byte
[7]	0	byte
[8]	0	byte
[9]	63	byte
[10]	0	byte
[11]	0	byte
[12]	0	byte
[13]	0	byte

Name
AvatarFrom
Microsoft.X
mscorlib.dll
mscorlib.dll

Figure 10.2 Description property in the debugger

Note

Xbox Live Indie Games prohibits utilizing avatar clothing and other assets that have been unlocked or purchased through the avatar marketplace unless they are outfitted on the player's avatar. So if you are creating a custom avatar for a nonplayer character, you are restricted in using only the built-in avatar assets and can't use items you purchased from the avatar marketplace.

The Changed Event

`AvatarDescription` provides an event called `Changed`. This event can fire while your game is running in the event that the user has updated his or her avatar.

Loading Avatar Animations with AvatarAnimation

There are a number of built-in animations that enable you to quickly animate an avatar in your game. The animations that are provided were created for use in the Xbox dashboard and consist of idle animations along with some expressive animations like celebrate and clap.

To add an animation to your game, the first thing you need is a new member variable in your game to hold the animation. Add the following member variable to your game:

```
AvatarAnimation avatarAnimation;
```

Next, in the game's `LoadContent` method, you can load the following celebrate animation:

```
// Load celebrate animation
avatarAnimation = new AvatarAnimation(AvatarAnimationPreset.Celebrate);
```

The `AvatarAnimation` constructor takes an `AvatarAnimationPreset` enumeration to specify which animation to load. Table 10.1 shows all of the built-in animation types that are defined by the `AvatarAnimationPreset` enumeration.

Table 10.1 **AvatarAnimationPreset Enumerations**

`Celebrate`
`Clap`
`FemaleAngry`
`FemaleConfused`
`FemaleCry`
`FemaleIdleCheckNails`
`FemaleIdleFixShoe`
`FemaleIdleLookAround`
`FemaleIdleShiftWeight`
`FemaleLaugh`
`FemaleShocked`
`FemaleShocked`
`FemaleYawn`
`MaleAngry`
`MaleConfused`
`MaleCry`
`MaleIdleCheckHand`
`MaleIdleLookAround`
`MaleIdleShiftWeight`
`MaleIdleStretch`
`MaleLaugh`
`MaleSurprised`
`MaleYawn`
`Stand0`
`Stand1`
`Stand2`
`Stand3`
`Stand4`
`Stand5`
`Stand6`
`Stand7`
`Wave`

Updating the Animation

There are two methods that you can use to update the current playback position of the animation. The first and easiest way is to use the `Update` method. The `Update` method takes a `TimeSpan` for the current frame's elapsed time and a boolean value that indicates whether the animation should be looped when it reaches the end of the animation.

To update the animation, add the following to your game's `Update` method:

```
// Update the animation with the elapsed frame time
// Set the looping parameter to true
avatarAnimation.Update(gameTime.ElapsedGameTime, true);
```

The `gameTime.ElapsedGameTime TimeSpan` is used for the current frame's length. This updates the animation in sync with how fast your game is updating. The value of `true` is passed for the loop argument so that the animation loops when it reaches the end of the animation.

The second method for updating the avatar's playback location is to set the value explicitly. The `AvatarAnimation CurrentPosition` property is a `TimeSpan` value that represents the amount of time from the start of the animation. This value can be set to any valid range from 0 to the end length of the animation. The length of the animation can be retrieved by using the `Length` property of the animation.

Transforms and Expression of the Animation

The animation returns two sets of values for the current playback position. The first is the location for each of the bones in the avatar skeleton. The `BoneTransforms` property returns a `ReadOnlyCollection` of `Matrix` values for each of the bone positions and orientations.

The second set of animation values are for the textures to use for the avatar's face. Although the body uses skeletal animation to transform the vertices of the avatar, the face is made up of textures for the eyes, eyebrows, and mouth. The `AvatarAnimation` provides the Expression property to return an `AvatarAnimation` structure.

The `AvatarAnimation` structure provides properties for the `LeftEye`, `RightEye`, `LeftEyebrow`, `RightEyebrow`, and `Mouth`. Each of these represents the texture to use for the corresponding facial feature.

Table 10.2 shows the `AvatarEye` enumeration values.

Table 10.2 **AvatarEye Enumerations**

`Angry`	`LookRight`
`Blink`	`LookUp`
`Confused`	`Neutral`
`Happy`	`Sad`
`Laughing`	`Shocked`
`LookDown`	`Sleeping`
`LookLeft`	`Yawning`

Table 10.3 shows the `AvatarEyebrow` enumeration values.

Table 10.3 **AvatarEyebrow Enumerations**

`Angry`	`Raised`
`Confused`	`Sad`
`Neutral`	

Table 10.4 shows the `AvatarMouth` enumeration values.

Table 10.4 **AvatarMouth Enumerations**

`Angry`	`PhoneticEe`
`Confused`	`PhoneticFv`
`Happy`	`PhoneticL`
`Laughing`	`PhoneticO`
`Neutral`	`PhoneticW`
`PhoneticAi`	`Sad`
`PhoneticDth`	`Shocked`

Avatar Animation Functionality Though an Interface

Although the built-in avatar animations are convenient, you might want to provide your own custom animations or to programmatically generate animations using IK. The `IAvatarAnimation` interface is provided to enable you to create your own custom avatar playback systems that work with the avatar system.

The `IAvatarAnimation` interface exposes four properties: `BoneTransforms`, `CurrentPosition`, `Expression`, and `Length`. It also exposes one method: `Update`. The built-in `AvatarAnimation` type implements the `IAvatarAnimation` interface so you can combine your animation playback system with the built-in animations.

You learn more about implanting a custom animation playback system that uses the `IAvatarAnimation` interface later in this chapter.

Drawing the Avatar Using AvatarRenderer

Now that you have loaded the `AvatarDescription` and have an `AvatarAnimation` to play, you just need models and textures for the avatar and draw them to the screen. This is precisely what the `AvatarRenderer` type does.

To draw the avatar in the game, you need to add a member variable to store the `AvatarRenderer`. Add the following member variables to your game class:

```
AvatarRenderer avatarRenderer;

Matrix world;
Matrix view;
Matrix projection;
```

In the game's `LoadContent` method, create an instance of the `AvatarRenderer` from one of the previous `AvatarDescription` instances you created previously.

```
// Load avatar renderer for random description
avatarRenderer = new AvatarRenderer(avatarDescription);

// Rotate avatar to face the screen
world = Matrix.CreateRotationY(MathHelper.Pi);
// Set view and projection matrices to something sensible
view = Matrix.CreateLookAt(new Vector3(0, 1, 3), new Vector3(0, 1, 0),
➥Vector3.Up);
projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                         GraphicsDevice.Viewport.AspectRatio,
                                         0.01f, 200.0f);
```

After creating a new instance of the `AvatarRenderer`, set default values for the `world`, `view`, and `projection` matrices. By default, the avatar faces in the negative Z direction, so set the world matrix to rotate around the Y axis.

If you are using the asynchronous `AvatarDescription.BeginGetFromGamer` method, then you can load the `AvatarRenderer` in the `AsyncCallback` method that you pass to `BeginGetFromGamer`.

The last bit of code is to draw the avatar. In your game's `Draw` method add the following lines of code:

```
// Set the world, view, and projection matrices
avatarRenderer.World = world;
avatarRenderer.View = view;
avatarRenderer.Projection = projection;

// Draw the avatar to the current render target
avatarRenderer.Draw(avatarAnimation);
```

Before you draw, first set the `World`, `View`, and `Projection` matrices on the `AvatarRenderer` instance. You can simply call the `Draw` method passing in any `AvatarAnimation` or other type that `implements` the IAvatarAnimation interface.

Running the sample should look like Figure 10.3.

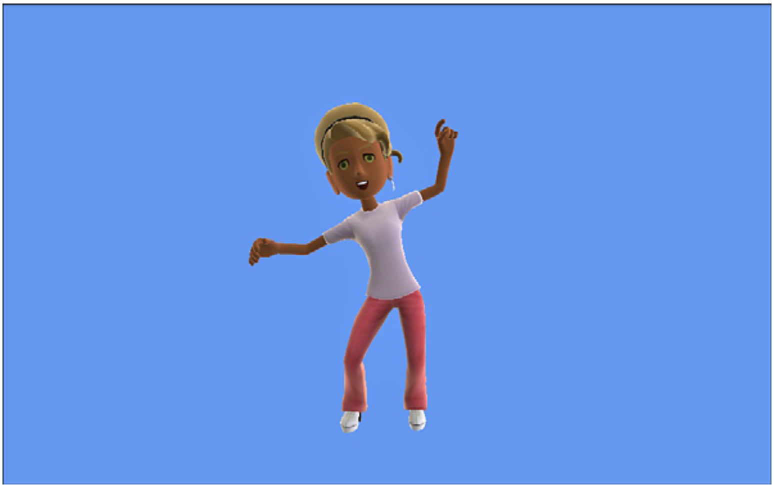


Figure 10.3 Avatar rendering animated with the celebrate animation

Drawing While Loading an Avatar

Although the `AvatarRenderer` constructor is not asynchronous and it returns immediately the loading, the geometry and textures that make up the avatar can take a few frames. During that time, if you call the `Draw` method, notice an animated swirling pattern like that shown in Figure 10.4.

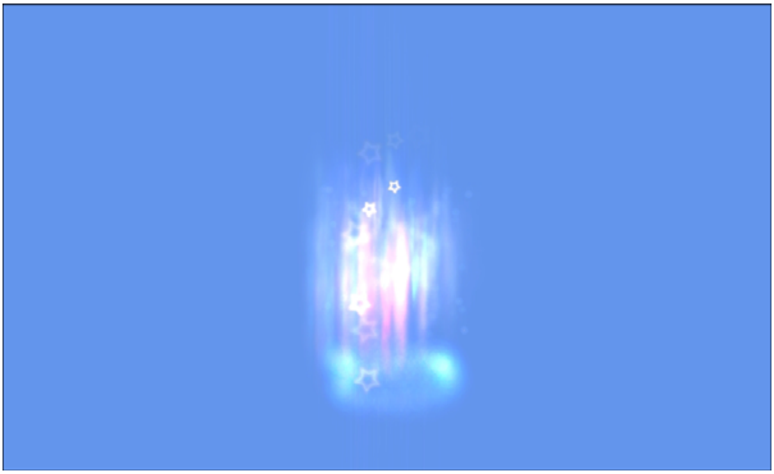


Figure 10.4 Avatar loading effect

If you don't want to display the loading effect, you can use the overload of the `AvatarRenderer` constructor that takes a boolean value for the `useLoadingEffect` parameter. Passing in false causes the loading effect to not display while the avatar is loading.

Deterring the Current State of the AvatarRenderer

At times, you might want to query the state of the `AvatarRenderer` to see whether it has completed loading. The `State` property can be used to determine the current state of an `AvatarRenderer` instance. The return value is an `AvatarRendererState` enumeration that can have three values of `Loading`, `Ready`, and `Unavailable`. The `Loading` and `Ready` states are self-explanatory. The `Unavailable` state is a special case that occurs in a couple of situations. The first is on `Windows` when any `AvatarRenderer` instance returns `Unavailable` because avatars can't be rendered on `Windows`. The second situation is when the models and textures that make up the avatar are corrupted on the Xbox and can't be downloaded.

> **Note**
>
> When avatars were first introduced, their proportions were not close to those of a real person. For example, their arms were too short. To help support player mapping between their avatar and their body using Kinect, the proportions of the avatars were updated to better match those of a human. The new avatars are referred to as V2 avatars. XNA Game Studio 4.0 utilizes V2 avatars.

Modifying Avatar Lighting

Now that you have been able to load and draw some avatars, it's time to dig a little deeper so they can better integrate into your game. Your game might take place at night or use some special colors in the lighting of your game. The `AvatarRenderer` provides the capability to set the ambient light color and one directional light.

Let's modify the existing sample to update the lighting to use when drawing the avatar. Add the following code to your game's `Draw` method before calling the `AvatarRenderer.Draw` method.

```
// Set the avatar lighting values
Vector3 lightDirection = new Vector3(1, 1, 0);
lightDirection.Normalize();
avatarRenderer.LightDirection = lightDirection;
avatarRenderer.LightColor = new Vector3(0, 1, 0);
avatarRenderer.AmbientLightColor = new Vector3(0.25f, 0.25f, 0.25f)
```

First, create a new light direction, which is pointing in the right and up direction. The direction needs to be normalized before setting the `LightDirection` property on the `AvatarRenderer`.

Set the color of the directional light to fully green by using the `LightColor` property.

Finally, set the ambient lighting value to around 25 percent of each color channel by using the `AmbientLightColor` property.

Running the sample now shows a green light on the avatar's lower left. It should look similar to Figure 10.5.

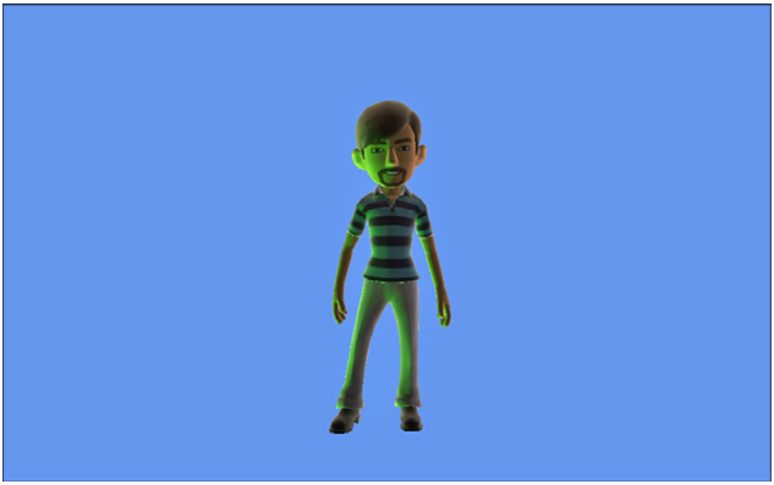


Figure 10.5 Avatar with modified lighting

Playing Multiple Animations

Although we have been concentrating on the built-in animations, it is possible to customize the animation. One example might be that you want to play a walking animation on the lower half of the avatar while the upper arms of the avatar throw a baseball.

To use two different animations for different portions of the avatar skeleton, identify which bones you want to update with which animation.

Let's update the existing sample to play back two animations at the same time. Use the celebrate animation for most of the avatar and play the wave animation on the avatar's right arm.

You need an additional `AvatarAnimation` and some additional member variables in your game. Add the following member variables to your game:

```
AvatarAnimation avatarAnimationCelebrate;
AvatarAnimation avatarAnimationWave;

// Combined transforms for the two animations
List<Matrix> combinedBoneTransfroms = new List<Matrix>(AvatarRenderer.BoneCount);

// Bone index values for the right arm
List<int> rightArmBones;
```

Along with the two `AvatarAnimation` variables for the celebrate and wave animations, you need two additional variables and a `List` of matrices that will store the avatar skeleton bone positions. The number of bones is defined by `BoneCount` constant field. You also need a list of index values for the bones that make up the right arm so you know which index values to update with the wave animation.

Next, in the games `LoadContent` method, add the following lines of code:

```
// Load celebrate animation
avatarAnimationCelebrate = new AvatarAnimation(AvatarAnimationPreset.Celebrate);
// Load the wave animation
avatarAnimationWave = new AvatarAnimation(AvatarAnimationPreset.Wave);

// Find all of the children of the right shoulder
rightArmBones = FindChildrenBones(AvatarBone.ShoulderRight,avatarRenderer.
➥ParentBones);

// Populate the list of bone transforms
for (int i = 0; i < AvatarRenderer.BoneCount; ++i)
{
    combinedBoneTransfroms.Add(Matrix.Identity);
}
```

The first two lines load the celebrate and wave animations. Populate the `rightArmBones` List by calling a helper method `FindChildrenBones`, which you create next. This helper method finds all of the children for a given `AvatarBone`. The last portion of code adds some default values to the `combinedBoneTransfroms` List that you use to draw the avatar.

Create the `FindChildrenBones` method that returns all of the children for a given `AvatarBone`.

```
private List<int> FindChildrenBones(AvatarBone parentBone,
                                    ReadOnlyCollection<int> parentBones)
{
    List<int> children = new List<int>();
    children.Add((int)parentBone);

    // Start search one past the parent
    int currentBone = (int)parentBone + 1;

    // Loop over all of the bones
    while (currentBone < parentBones.Count)
    {
        // See if the bone has a parent in the list
        if (children.Contains(parentBones[currentBone]))
        {
            children.Add(currentBone);
        }
        currentBone++;
    }

    return children;
}
```

To find all of the children for the given `parentBone`, you need the collection of `parentBones` from the `AvatarRenderer`. The `ParentBones` property returns a collection of index values that represent the parent for each index location. For example, the `AvatarBone.Neck` enumeration has an integer value of 14. If you look up the fourteenth item in the `parentBones` collection, you get an integer value of 5, which is the integer value for the `AvatarBone.BackUpper` enumeration because the upper back is the parent of the neck.

The avatar skeleton is sorted by depth so the lower index values represent bones that are closer to the root. This also means that you can loop over the list of bones and know that the parent of the current bone has already been processed. This is useful in cases where you want to add all of the parent bones for a given bone. As you loop over all of the bones, check whether the parent index value is already in the list. If it is, then the current bone must be a child of one of the parent bones already in the list.

For each frame, you should update both animations and copy the bone positions to the `combinedBoneTransfroms`. In your game's `Update` method, add the following lines of code:

```
// Update both animations
avatarAnimationCelebrate.Update(gameTime.ElapsedGameTime, true);
avatarAnimationWave.Update(gameTime.ElapsedGameTime, true);

// Copy the celebrate transforms
for (int i = 0; i < combinedBoneTransfroms.Count; i++)
```

```
{
    combinedBoneTransfroms[i] = avatarAnimationCelebrate.BoneTransforms[i];
}

// Overwrite the transforms for the right arm using the values
// from the wave animation
for (int i = 0; i < rightArmBones.Count; i++)
{
    combinedBoneTransfroms[rightArmBones[i]] =
                                    avatarAnimationWave.BoneTransforms[right
                                    ArmBones[i]];
}
```

First, call each animation's `Update` method passing in the frame's `ElapsedGameTime` and a loop value of `true`. Copy all of the `BoneTransfroms` from the celebrate animation. Then, loop all of the `rightArmBones` index values and copy the bone values from the wave animation into the `combinedBoneTransfroms` list.

The final change is to update how the `AvatarRender` draw call is called. In the games `Draw` method, update the following lines of code:

```
// Draw the avatar using the combined bone transforms
avatarRenderer.Draw(combinedBoneTransfroms, avatarAnimationCelebrate.Expression);
```

In this overload of the AvatarRenderer Draw method, it takes a list of the bone transforms and an AvatarExpression. Use the Expression from the celebrate animation and your `combinedBoneTransfroms` list.

Running the sample now should display the avatar playing the celebrate animation except for the right arm, which is waving as shown in Figure 10.6 below.

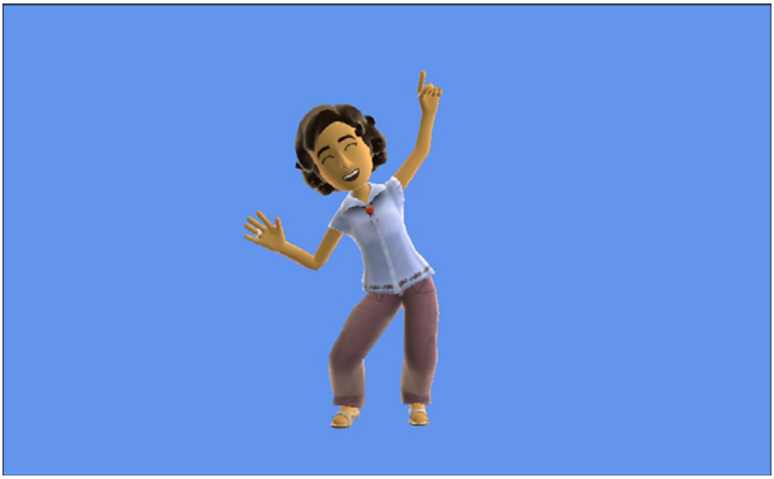


Figure 10.6 Avatar playing the combined celebrate and wave animation

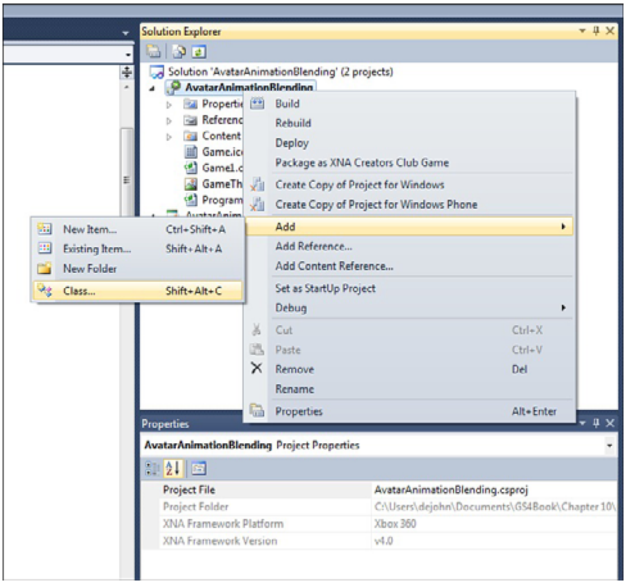
Blending Between Animations

Your game will most likely contain more than one animation. If you switch between animations, the location of the bones of the avatar quickly move to their new position and appear to pop from one location to another. What you really want is a way to blend from one animation to another smoothly so that the avatar does not appear to pop from one to the other.

To accomplish this, blend from using one set of transforms to the other over a short period of time. The bone's position and rotation values need to be interpolated over the time from one animation to the other.

To demonstrate how to blend between two avatar animations, create a new type that you will use to play back the blending avatar animation.

Add a new class to the sample by right-clicking the project and selecting Add -> Class as seen in Figure 10.7.


Solution Explorer
Solution 'AvatarAnimationBlending' (2 projects)
AvatarAnimationBlending
Properties
References
Content
Game.ico
Game1.cs
GameThumbnail
Program
Build
Rebuild
Deploy
Package as XNA Creators Club Game
Create Copy of Project for Windows
Create Copy of Project for Windows Phone
Add
Add Reference...
Add Content Reference...
Set as StartUp Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Properties Alt+Enter
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Class... Shift+Alt+C
Properties
AvatarAnimationBlending Project Properties

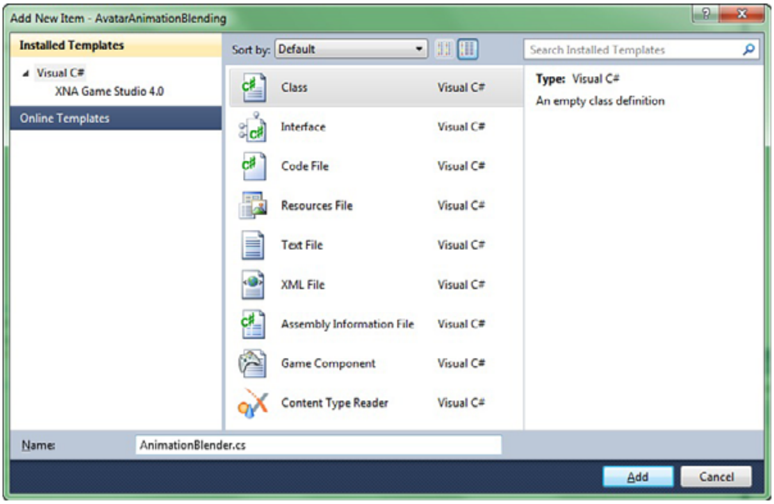
Project File AvatarAnimationBlending.csproj
Project Folder C:\Users\dejohn\Documents\GS4Book\Chapter 10\
XNA Framework Platform Xbox 360
XNA Framework Version v4.0

Figure 10.7 Add new class menu

When the Add New Item dialog appears, enter the name of the new class called `AnimationBlender.cs` and select Add as in Figure 10.8.

Our new class needs to be public and implement the `IAvatarAnimation` interface. Update the class definition for your new class to the following:

```
public class AnimationBlender : IAvatarAnimation
```


Add New Item - AvatarAnimationBlending
Installed Templates
Visual C#
XNA Game Studio 4.0
Online Templates
Search Installed Templates
Class Visual C#
Interface Visual C#
Code File Visual C#
Resources File Visual C#
Text File Visual C#
XML File Visual C#
Assembly Information File Visual C#
Game Component Visual C#
Content Type Reader Visual C#
Add
Cancel

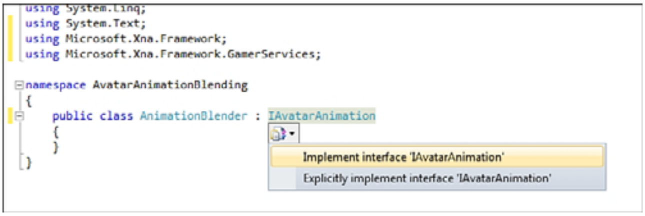
Sort by: Default

Type: Visual C#
An empty class definition

Name: AnimationBlender.cs

Figure 10.8 Add New Item dialog

After you type the interface name, Visual Studio displays a tool tip that can be used to implement the interface with stubs as shown in Figure 10.9.



```
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.GamerServices;

namespace AvatarAnimationBlending
{
    public class AnimationBlender : IAvatarAnimation
    {
    }
}
```

Implement interface 'IAvatarAnimation'
Explicitly implement interface 'IAvatarAnimation'

Figure 10.9 Tool tip for interface implementation

The `AnimationBlender` contains a two-member variable to hold references to the current and next animation that are blended together. There are two states the `AnimationBlender` can be in. The first is playing just a single animation, which works just like a normal `AvatarAnimation`. The second is a blending state where the current animation starts to blend to the next animation and ends when the blend is complete and the next animation becomes the current animation.

Add the following member variables to the new `AnimationBlender` class:

```
// The current aniamtion that is playing
private AvatarAnimation currentAnimation;
// The animation to blend to
private AvatarAnimation nextAnimation;
```

```
// The total time to blend to the next animation
private TimeSpan totalBlendTime = TimeSpan.FromMilliseconds(300);
// How far along the blend we are
private TimeSpan currentBlendTime;
```

We also need two variables to store the current blend time and how long the blend from one animation to the other should take.

Implement each of the properties exposed by the `IAvatarAnimation` interface. The first is the `BoneTransfroms`.

```
/// <summary>
/// Returns the bone transforms to use with the AvatarRenderer
/// </summary>
public ReadOnlyCollection<Matrix> BoneTransforms
{
    get { return boneTransforms; }
}
private Matrix[] blendedAvatarBones = new Matrix[AvatarRenderer.BoneCount];
private ReadOnlyCollection<Matrix> boneTransforms;
```

Next, implement the `CurrentPosition` property.

```
/// <summary>
/// The current position in the animation
/// If we are blending then we use the next animation
/// </summary>
public TimeSpan CurrentPosition
{
    get
    {
        TimeSpan returnValue;

        if (nextAnimation != null)
            returnValue = nextAnimation.CurrentPosition;
        else
            returnValue = currentAnimation.CurrentPosition;

        // Call update to set the avatar transforms
        Update(TimeSpan.Zero, false);

        return returnValue;
    }
    set
    {
        if (nextAnimation != null)
            nextAnimation.CurrentPosition = value;
        else
            currentAnimation.CurrentPosition = value;
    }
}
```

For the `CurrentPosition`, use the value from the `nextAnimation` if you are currently blending; otherwise, use the `currentAnimation` value. Do the same for the `Expression` and `Length` properties.

```
/// <summary>
/// The current expression in the animation
/// If we are blending then we use the next animation
/// The expression is not blended so we just use
/// the next animations expression
/// </summary>
public AvatarExpression Expression
{
    get
    {
        if (nextAnimation != null)
            return nextAnimation.Expression;
        else
            return currentAnimation.Expression;
    }
}

/// <summary>
/// The length of the animation
/// If we are blending then we use the next animation
/// </summary>
public TimeSpan Length
{
    get
    {
        if (nextAnimation != null)
            return nextAnimation.Length;
        else
            return currentAnimation.Length;
    }
}
```

The constructor is quite simple and takes the initial `AvatarAnimation` that should be playing. Add the following constructor to the `AnimationBlender`:

```
public AnimationBlender(AvatarAnimation startingAnimation)
{
    currentAnimation = startingAnimation;

    boneTransforms = new ReadOnlyCollection<Matrix>(blendedAvatarBones);
}
```

Set the `currentAnimation` variable and initialize the `ReadOnlyCollection` of matrices that you return from the `BoneTransforms` property.

Now you need an animation to call when you start playing a new animation. This kicks off the blend from the current animation to the new animation. Add the following method to the `AnimationBlender` class:

```
/// <summary>
/// Starts the process of blending to the next animation
/// </summary>
public void Play(AvatarAnimation animation)
{
    // Animation to blend to
    nextAnimation = animation;
    // Reset the animation to start at the beginning
    nextAnimation.CurrentPosition = TimeSpan.Zero;

    // Set our current blend position
    currentBlendTime = TimeSpan.Zero;
}
```

First, set the `nextAnimation` to the one passed into the method. Then, reset the animation's `CurrentPosition` so that it is at the start of the animation. This is not required, but is helpful when playing the same animations over and over again. The last line of code resets the `currentBlendTime` back to zero.

The final method that is required for your `AnimationBlender` class is to implement the `Update` method. The `Update` method is where you update the private bone transforms and perform any blending that is required. Add the following `Update` method to the `AnimationBlender` class:

```
/// <summary>
/// Updates the animation and performs blending to the next animation
/// </summary>
public void Update(TimeSpan elapsedAnimationTime, bool loop)
{
    // Update the current animation
    currentAnimation.Update(elapsedAnimationTime, loop);

    // If we are not blending to an animation just copy the current transforms
    if (nextAnimation == null)
    {
        currentAnimation.BoneTransforms.CopyTo(blendedAvatarBones, 0);
        return;
    }

    // If we get here we are blending and we need to update the next animation
    nextAnimation.Update(elapsedAnimationTime, loop);

    // Update where we are in the blend
    currentBlendTime += elapsedAnimationTime;
```

```
        // Calculate blend factor
        float blendAmount = (float)(currentBlendTime.TotalSeconds /
                                    totalBlendTime.TotalSeconds);

        // If the blend is over 1 then we are done blending
        // We can just use the next animation transforms
        if (blendAmount >= 1.0f)
        {
            // Set the current animtion and remove the target animation
            currentAnimation = nextAnimation;
            nextAnimation.BoneTransforms.CopyTo(blendedAvatarBones, 0);
            nextAnimation = null;
            return;
        }

        // We need to store the rotation and translation for each bone in
        // the current and next animations. We then need to calculate the
        // final values using the blend amount as a weight between the
        // current and next values.
        Quaternion currentRotation, nextRotation, blendedRotation;
        Vector3 currentTranslation, nextTranslation, blendedTranslation;

        // Loop all bone transforms
        for (int i = 0; i < blendedAvatarBones.Length; ++i)
        {
            currentRotation =
              Quaternion.CreateFromRotationMatrix(currentAnimation.BoneTransforms[i]);
            currentTranslation = currentAnimation.BoneTransforms[i].Translation;
            nextRotation =
                 Quaternion.CreateFromRotationMatrix(nextAnimation.BoneTransforms[i]);
            nextTranslation = nextAnimation.BoneTransforms[i].Translation;

            // Calculate the blended rotation
            Quaternion.Slerp(ref currentRotation, ref nextRotation,
                             blendAmount, out blendedRotation);

            // Calculate the blended translation
            Vector3.Lerp(ref currentTranslation, ref nextTranslation,
                         blendAmount, out blendedTranslation);

            // Create the blended bone matrix
            blendedAvatarBones[i] = Matrix.CreateFromQuaternion(blendedRotation) *
                                    Matrix.CreateTranslation(blendedTranslation);
        }
    }
```

That was a lot code, so let's take a look at it piece by piece. The first section of code updates the current animation and checks whether there is blending by checking the `nextAnimation` for null. If there is no blending, the transforms from the current animation are copied to the `blendedAvatarBones`.

If there is blending, then the `nextAnimation Update` method is called to update the playback of the animation. The amount of time the blending has been occurring is then updated and the `blendAmount` is calculated. The `blendAmount` ranges from 0 to 1 over the course of the blend. If the value is equal to 1 or higher, then you should complete the blend and set the `currentAnimation` to the `nextAnimation`.

The final section of code is the blend between the two sets of bones from the animations. Each bone in the avatar skeleton is looped over and the matrix value for the animation frame is broken down into its rotation and translation components. The rotation value is then calculated by taking the `Slerp` value using the `blendAmount`. The translation value is then calculated with a linear transformation using `Lerp`. The rotation and translation are then combined to form the blended bone transform that is saved in the `blendedAvatarBones` array.

Now that you created the `AnimationBlender` class, you can use it in the ongoing sample. The game needs an array of `AvatarAnimations` and an instance of the `AnimationBlender`. You also need to save the state of the `GamePad`, so you can track button presses. Add the following member variables to your game class:

```
AvatarAnimation[] avatarAnimations;
AnimationBlender animationBlender;

GamePadState lastGamePadState;
```

In the game's `LoadContent` method, load the multiple animations that you will use and initialize your `AnimationBlender`. Add the following lines of code to the `LoadContent` method:

```
// Load all of the animation
avatarAnimations = new AvatarAnimation[3];
avatarAnimations[0] = new AvatarAnimation(AvatarAnimationPreset.Celebrate);
avatarAnimations[1] = new AvatarAnimation(AvatarAnimationPreset.Clap);
avatarAnimations[2] = new AvatarAnimation(AvatarAnimationPreset.Wave);

// Create the animation blender
animationBlender = new AnimationBlender(avatarAnimations[0]);
```

Next, in the game's `Update` method, call `Update` on the `AnimationBlender`. Also detect button presses and play different animations depending on which button you press.

```
// Update the animation blender
animationBlender.Update(gameTime.ElapsedGameTime, true);

// Get the current state of player one controller
GamePadState currentGamePadState = GamePad.GetState(PlayerIndex.One);
```

```
// If the user presses the X, Y, or B buttons blend to an animation
if (currentGamePadState.Buttons.X == ButtonState.Pressed &&
    lastGamePadState.Buttons.X == ButtonState.Released)
{
    animationBlender.Play(avatarAnimations[0]);
}
if (currentGamePadState.Buttons.Y == ButtonState.Pressed &&
    lastGamePadState.Buttons.Y == ButtonState.Released)
{
    animationBlender.Play(avatarAnimations[1]);
}
if (currentGamePadState.Buttons.B == ButtonState.Pressed &&
    lastGamePadState.Buttons.B == ButtonState.Released)
{
    animationBlender.Play(avatarAnimations[2]);
}

// Save the state so we can prevent repeat button presses
lastGamePadState = currentGamePadState;
```

The final step is to use the `AvatarBlender` to draw the avatar. `Update` the `Draw` method with the following line to draw the avatar using the blended animation:

```
avatarRenderer.Draw(animationBlender);
```

Running the sample now should enable you to press the X,Y, and B buttons to blend between animations. The blended animation should look similar to Figure 10.10 below:

Interacting with Objects

Sometimes you want the avatar to interact with items in your game. If you have a baseball game, you might want the avatars to be able to hold a baseball bat and throw a baseball. The avatar bones returned from the avatar animations and the values expected by the `AvatarRenderer.Draw` method are in local bone space. This means that the bones are relative to their parent bone and the bind pose for the avatar. To find the location of the avatar's hand to place a baseball, find the world space position of the bone that represents the avatar's hand. Then, use this bone to place the baseball model.

Let's see how this works by creating an example of an avatar lifting a hand weight. Start with the previous code from this chapter that draws a random avatar.

For additional member variables, add the following member variables.

```
 // Avatar bones in world space
List<Matrix> bonesWorldSpace;

Model handWeight;
Matrix handWorldMatrix;
```

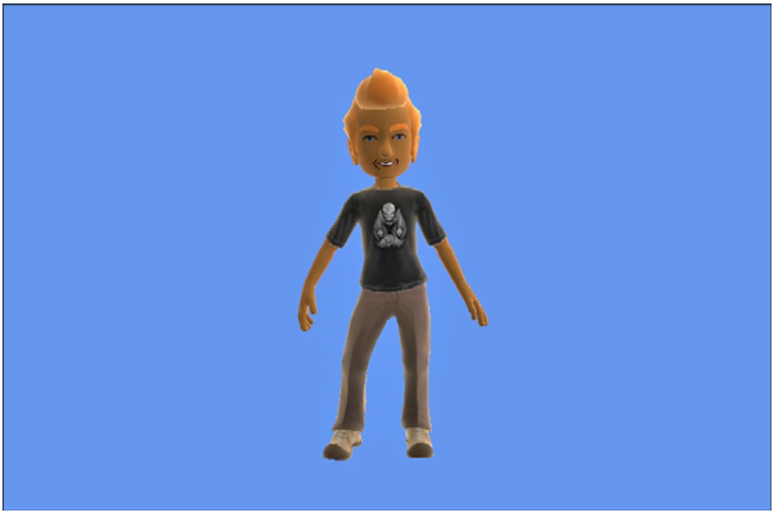


Figure 10.10 Avatar drawn with a blended animation

The `bonesWorldSpace` stores the world space values for each of the bones that make up the avatar skeleton. You also need variables for the hand-weight model and the world matrix used to draw it.

In the `LoadContent` method, load the model and populate the list of bones in world space. Add the following lines of code to your game's `LoadContent` method:

```
// Load the hand weight model
handWeight = Content.Load<Model>("HandWeight");

// Turn on default lighting
foreach (ModelMesh mesh in handWeight.Meshes)
{
    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();
    }
}

// Populate the world space bone list
bonesWorldSpace = new List<Matrix>(AvatarRenderer.BoneCount);
for (int i = 0; i < AvatarRenderer.BoneCount; i++)
    bonesWorldSpace.Add(Matrix.Identity);
```

Next, in the game's `Update` method, calculate the world space transforms for the avatar animation. Add the following to your game's `Update` method:

```
if (avatarRenderer.State == AvatarRendererState.Ready)
{
    ConvertBonesToWorldSpace(avatarRenderer, avatarAnimation);
}

// Find the position of the right hand bone
handWorldMatrix = bonesWorldSpace[(int)AvatarBone.SpecialRight];
```

Before you convert the bone transforms into world space, check that the `AvatarRenderer.State` property is set to `Ready` so you can verify that the avatar and its skeleton have completed loading. Then, call the `ConvertBonesToWorldSpace` method passing in the `AvatarRenderer` and `AvatarAnimation` we want to find the world space position of. After calculating the world space transforms, use the `AvatarBone.SpecialRight` bones transform for the world matrix for the hand-weight model. The `SpecialRight` bone is a piece of the avatar skeleton that is located near the inside of the forearm in front of the hand. It is a good bone to use when you want to place objects in the avatar's hand.

To implement the `ConvertBonesToWorldSpace` helper method, add the following method to your game:

```
/// <summary>
/// Convert the avatar bones into world space
/// </summary>
private void ConvertBonesToWorldSpace(AvatarRenderer renderer, AvatarAnimation
➥animation)
{
    // Bind pose of the avatar skeleton
    // Values are in local space and relative to the parent bone
    IList<Matrix> bindPose = renderer.BindPose;
    // Animation bone transforms are in local space and relative to the parent bone
    IList<Matrix> animationPose = animation.BoneTransforms;
    // Parent index values for each of the bones
    IList<int> parentIndexes = renderer.ParentBones;

    // Transform each bone into world space
    for (int i = 0; i < AvatarRenderer.BoneCount; i++)
    {
        // Partent world space matrix
        // The world matrix of the avatar renderer is used for the root
        Matrix parentMatrix = (parentIndexes[i] != -1)
                            ? bonesWorldSpace[parentIndexes[i]]
                            : renderer.World;

        // Calculate world space matrix for the bone
        bonesWorldSpace[i] = Matrix.Multiply(Matrix.Multiply(animationPose[i],
                                                              bindPose[i]),
                                                              parentMatrix);
    }
}
```

To calculate the bone's position in world space, you first need the bind pose of the avatar. The animation frames are relative to the bind pose transforms, so you need both the animation transforms and the bind pose transforms. Each of the bones is also relative to its parent bone, so you also need to multiply the transform by its parent world space matrix. For the root node, use the World property of the `AvatarRenderer` because this is the location in world space of the avatar. Again, you can loop over the bones linearly as you calculate the world space transforms because you know the parent bones are calculated before any of the children.

Finally, you can draw the hand-weight model using the world space transform you calculated. Add the following lines of code to your game's `Draw` method:

```
// Draw the hand weight model
handWeight.Draw(handWorldMatrix, view, projection);
```

Running the sample displays an avatar with the weight model in his or her right hand (see Figure 10.11).

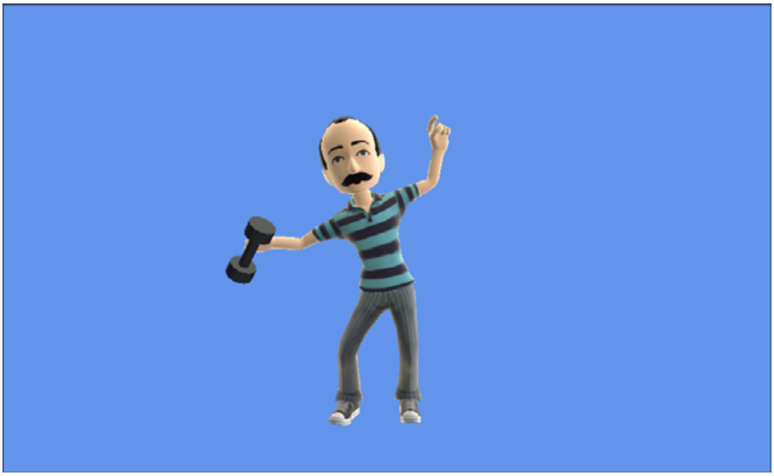


Figure 10.11 Hand weight drawn at the avatar's hand location

2D Avatars Using Render Targets

Although the avatars render in 3D, it is possible to use them within your 2D game easily. One method to do this is to draw the animating avatars to a render target and then use the texture from the render target as you would any other 2D texture in your game. You can even build sprite sheets from the avatar animations.

Let's create a sample that shows how to render an avatar to a `RenderTarget` and then display the texture as a sprite. Start with the existing sample that displays a random avatar animation. For an additional member variable to store the `RenderTarget`, add the following member variable to your game's class.

```
RenderTarget2D renderTarget;
```

To initialize the `RenderTaget` so it is ready to be set on the device, add the following to the `LoadContent` method in your game:

```
// Create new render target to draw the avatar to it
renderTarget = new RenderTarget2D(GraphicsDevice, 512, 512, false,
                                  SurfaceFormat.Color, DepthFormat.Depth16);
```

You also want to change the projection matrix to use an aspect ratio of 1 because the width and height of the texture are equal.

```
projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                                 1, 0.01f, 200.0f);
```

The last portion of code changes the way you render the avatar in the game's `Draw` method. Update the game's `Draw` method to contain the following code:

```
// Set the render target
GraphicsDevice.SetRenderTarget(renderTarget);

// Set the render target to transparent
GraphicsDevice.Clear(Color.Transparent);

// Set the world, view, and projection matrices
avatarRenderer.World = world;
avatarRenderer.View = view;
avatarRenderer.Projection = projection;

// Draw the avatar to the current render target
avatarRenderer.Draw(avatarAnimation);

// Set back to the backbuffer
GraphicsDevice.SetRenderTarget(null);

// Clear the backbuffer
GraphicsDevice.Clear(Color.CornflowerBlue);

// Draw the render target as a sprite
spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.NonPremultiplied);
spriteBatch.Draw(renderTarget, Vector2.Zero, Color.White);
spriteBatch.End();
```

First, set your render target and clear the render target to transparent. You clear to transparent to be able to composite the texture with other textures in your game. Then, draw the avatar as you would normally by setting the `World`, `View`, and `Projection` transforms before calling the `Draw` method. This draws the avatar to the render target.

The render target is then set to null to switch back to the backbuffer. The backbuffer is then cleared to `CornflowerBlue` before the `SpriteBatch` is used to draw the render target to the screen.

If you run the sample now, notice that the the texture with the avatar appears in the upper right-hand corner of the screen. Because the rest of the texture is transparent, it is difficult to tell you are using a render target. You can change the clear color when clearing the render target to show that you are in fact drawing the avatar to the screen using a texture that is created using a render target. Figure 10.12 shows the avatar drawn to the screen using a render target.

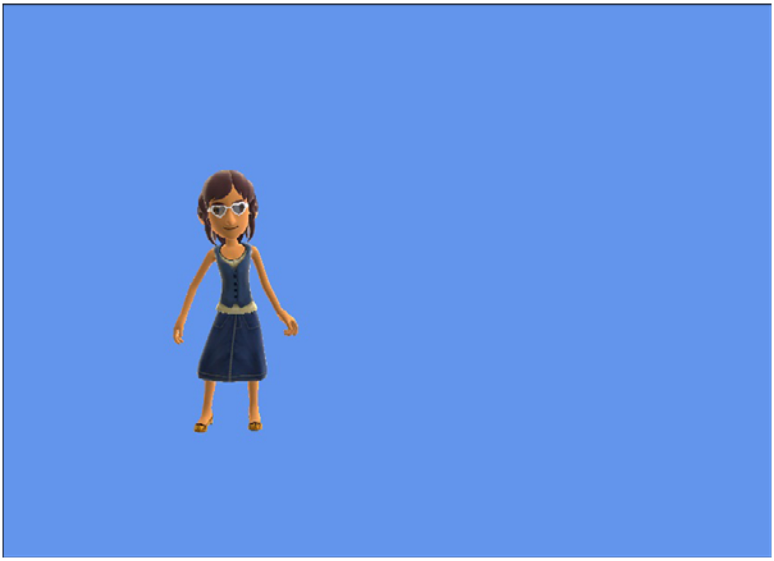


Figure 10.12 Avatar drawn to the screen using a render target

Custom Avatar Animations

Although the built-in animations presets provided by `AvatarAnimation` are a convenient way to animate the avatar, you will want to utilize custom animations in your game. If your game requires more than the avatar to just stand around, you need to use custom animations to do movements like walk or throw a ball.

Creating the Custom Animation

Custom animations require an artist to animate the avatar rig in a 3D content creation package. The avatar rig is available on the XNA Creators Club Online site at the following URL:

http://create.msdn.com/en-US/education/catalog/utility/avatar_animation_rig

The rig comes in a few versions for a number of different 3D content creation packages such as Maya, 3D Studio Max, and Softimage Mod Tool. After you download the rig, read the corresponding readme file contained within the download. This file instructs you on how to load the rig project file in the 3D content creation package of your choice. It is important that you follow the instructions in the readme file; otherwise, you might run into issues later when you are loading or exporting animations.

After you have the avatar rig loaded in a 3D content creation package of choice, you need to create the animation. Depending on the 3D content creation package you are using, this occurs in different ways. In most cases, the animator sets keyframes for the bone positions across a timeline. Please refer to documentation or tutorials for your specific 3D content creation package on how to animate a rigged character. Figure 10.13 shows the avatar rig loaded and animated using the Softimage Mod Tool.

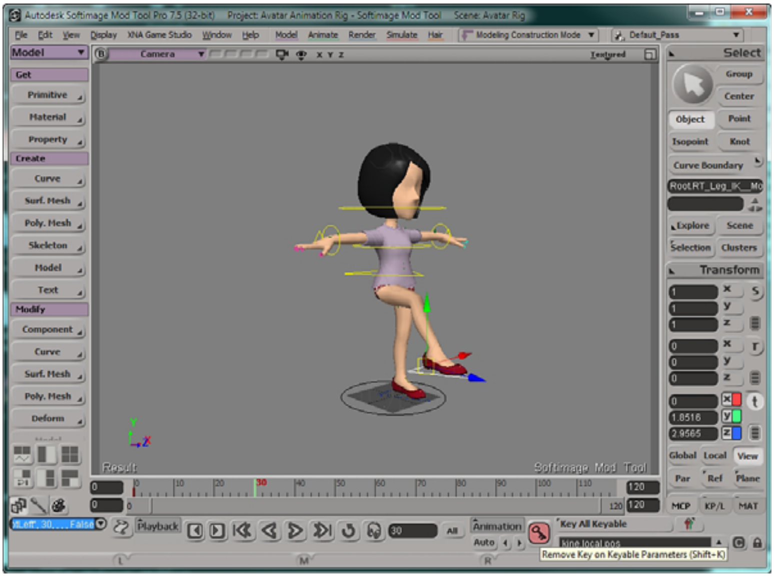

Autodesk Softimage Mod Tool Pro 7.5 (32-bit)   Project: Avatar Animation Rig - Softimage Mod Tool   Scene: Avatar Rig
File Edit View Display XNA Game Studio Window Help Model Animate Render Simulate Hair
Modeling Construction Mode
Default_Pass
Model
Get
Primitive
Material
Property
Create
Curve
Surf. Mesh
Poly. Mesh
Skeleton
Model
Text
Modify
Component
Curve
Surf. Mesh
Poly. Mesh
Deform
Camera
Textured
Select
Group
Center
Object
Point
Isopoint
Knot
Curve Boundary
RootRT_Leg_IK_Mo
Explore
Scene
Selection
Clusters
Transform
Global
Local
View
Par
Ref
Plane
MCP
KP/L
MAT
Result
Softimage Mod Tool
Playback
Animation
Auto
Key All Keyable
Remove Key on Keyable Parameters (Shift+K)

Figure 10.13 Avatar animation rig loaded into Softimage Mod Tool

After the animation is complete, it needs to be exported into an FBX file that you will load using a custom processor similar to how you loaded a skinned animation in Chapter

6, "Built-in Shader Effects." To export the animation, follow the instructions found in the readme file that was included in the rig download file.

There are a number of custom animations available on the XNA Creators Club Online site at the following URL:

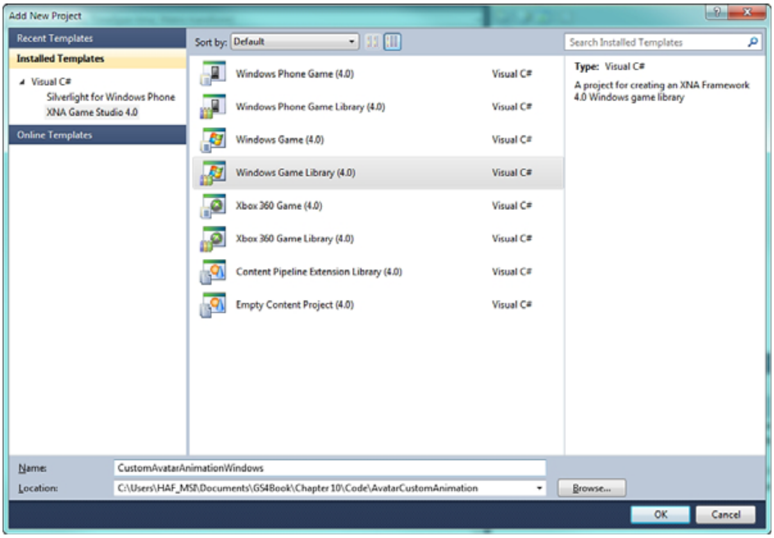
http://create.msdn.com/en-US/education/catalog/utility/avatar_animation_pack

Building the Custom Animation Type

The `AvatarRenderer` requires a set of bone transforms to set the positions of all the bones that make up the avatar each frame when rendering. When you load the custom animations, you will load a list of keyframes for each of the bones in the avatar. The custom animation uses the keyframes to determine which bone transforms to set at the current time in the animation.

Create three types that are used in both the content pipeline process you build and at runtime by your game. The first type you create stores the data associated with each keyframe. The second type stores the list of key fames and other data associated with an animation. The third type is the custom animation itself, which implements the `IAvatarAnimation` interface and can be used just like any other `AvatarAnimation` by your game.

Before you create new types, let's create the project that they will be built in. Right-click the solution and select Add -> New Project. In the Add New Project dialog, select the Windows Game Library project type and give the project the name `CustomAvatarAnimationWindows` and click the OK button as shown in Figure 10.14.


Add New Project
Recent Templates
Installed Templates
Visual C#
Silverlight for Windows Phone
XNA Game Studio 4.0
Online Templates
Sort by: Default
Search Installed Templates
Windows Phone Game (4.0) Visual C#
Windows Phone Game Library (4.0) Visual C#
Windows Game (4.0) Visual C#
Windows Game Library (4.0) Visual C#
Xbox 360 Game (4.0) Visual C#
Xbox 360 Game Library (4.0) Visual C#
Content Pipeline Extension Library (4.0) Visual C#
Empty Content Project (4.0) Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating an XNA Framework 4.0 Windows game library

Name: CustomAvatarAnimationWindows

Location: C:\Users\HAF_MSI\Documents\GS4Book\Chapter 10\Code\AvatarCustomAnimation

Figure 10.14 Add New Project dialog with a Windows Game Library selected

Now with your new project created, add your new types. Right-click the project and select Add -> Class. This creates a new C# code, which you should rename `Keyframe`. After renaming the file, double-click to open the file in the editor.

Although you created a new class, you really want the keyframe to be a structure. To do this, update the class definition to be a public structure like the following line:

```
public struct Keyframe
```

The keyframe structure is simple and contains only three members for the bone index of the avatar, the time when the keyframe occurs, and a matrix value of the transform for the bone at the time. Add the following lines of code to the keyframe structure:

```
// The index value of this bone in the avatar skeleton
public int BoneIndex;
// Time when this keyframe occurs in the animation
public TimeSpan Time;
// The bone transfrom
public Matrix Transform;

public Keyframe(int boneIndex, TimeSpan time, Matrix transform)
{
    BoneIndex = boneIndex;
    Time = time;
    Transform = transform;
}
```

Along with the three members for the bone index, time, and transform, you also include a helpful constructor that you will use when you create the keyframes in the content processor.

Now you need a type to hold the animation data that is saved out when you run the custom processor. Next create another new class. This time, rename the file `CustomAvatarAnimationData.cs` and add the following code:

```
// The animation data that is processed by the content pipeline
public class CustomAvatarAnimationData
{
    // The name of the animation
    [ContentSerializer]
    public string Name { get; private set; }

    // The total length the animation plays
    [ContentSerializer]
    public TimeSpan Length { get; private set; }

    // All of the keyframes that make up the animation
    [ContentSerializer]
    public List<Keyframe> Keyframes { get; private set; }
    // Parameterless constructor needed by the XNB deserializer
    private CustomAvatarAnimationData() { }

    public CustomAvatarAnimationData(string name, TimeSpan length,
                                     List<Keyframe> keyframes)
    {
        Name = name;
        Length = length;
        Keyframes = keyframes;
    }
}
```

The first property of the type is the `Name` of the animation. The next is a `TimeSpan` property that specifies the `Length` of the animation. The main portion of the animation data is the list of `Keyframe` objects. You also provide two constructors. The first is a parameter-less constructor, which is required to desterilize the object because the object needs to be constructed before the properties of the object are set. The second constructor is provided for coding convenience and sets the `Name`, `Length`, and `Keyframes` properties.

Now that you have the basic data that is stored in the custom animation class, you can create the actual custom animation type. Create a new class and change the name of the file to `AvatarCustomAnimation.cs`. The `AvatarCustomAnimation` class inherits from the `CustomAvatarAnimationData` type and implements the `IAvatarAnimation` interface. Update the class definition to look like the following:

```
public class AvatarCustomAnimation : CustomAvatarAnimationData, IAvatarAnimation
```

To implement the properties of the `IAvatarAnimation`, add the following properties to the `AvatarCustomAnimation` class:

```
public ReadOnlyCollection<Matrix> BoneTransforms
{
    get { return boneTransforms; }
}
private ReadOnlyCollection<Matrix> boneTransforms;
Matrix[] avatarBoneTransforms = new Matrix[AvatarRenderer.BoneCount];

public TimeSpan CurrentPosition
{
    get
    {
        return currentPosition;
    }
    set
    {
        currentPosition = value;
        // Reset the animation and update the current transfroms
        currentKeyframe = 0;
        Update(TimeSpan.Zero, false);
    }
}
private TimeSpan currentPosition = TimeSpan.Zero;
private int currentKeyframe = 0;

public AvatarExpression Expression
{
    get { return avatarExpression; }
}
AvatarExpression avatarExpression = new AvatarExpression();
```

Like the `AvatarAnimation`, the `BoneTransforms` and `Expression` are used by the `AvatarRenderer` to set the current bone positions and expression when drawing the avatar. The animation does not contain the expression data, so just set the expression to a new instance of `AvatarExpression`.

To add the constructor that initializes the custom avatar animation, add the following constructor to the `AvatarCustomAnimation`:

```
public AvatarCustomAnimation(CustomAvatarAnimationData animationData) :
    base(animationData.Name, animationData.Length, animationData.Keyframes)
{
    // Initialize bone transforms
    for (int i = 0; i < AvatarRenderer.BoneCount; i++)
    {
        avatarBoneTransforms[i] = Matrix.Identity;
    }
    boneTransforms = new ReadOnlyCollection<Matrix>(avatarBoneTransforms);

    // Populate the transfroms for the first frame
    Update(TimeSpan.Zero, false);
}
```

The constructor calls the base `CustomAvatarAnimationData` constructor to set the `Name`, `Length`, and `Keyframes`. Next the `avatarBoneTransforms` array is initialized to the `Identity`. Finally, call the `Update` method to have it set up the animation for playback in case the animation is used to render an avatar before the developer calls the `Update` method for the first time.

Add the following `Update` method to the `AvatarCustomAnimation` class:

```
public void Update(TimeSpan timeSpan, bool loop)
{
    // Incrament the curret time
    currentPosition += timeSpan;
    // Check current time against the length
    if (currentPosition > Length)
    {
        if (loop)
        {
            // Find the right time in the new loop iteration
            while (currentPosition > Length)
            {
                currentPosition -= Length;
            }
            // Set the keyframe to 0.
            currentKeyframe = 0;
        }
        else
        {
            // If the animation is not looping,
            // then set the time to the end of the animation.
            currentPosition = Length;
        }
    }
    // Check to see if we are less than zero
    else if (currentPosition < TimeSpan.Zero)
    {
        if (loop)
        {
            // If the animation is looping,
            // then find the right time in the new loop iteration
            while (currentPosition < TimeSpan.Zero)
            {
                currentPosition += Length;
            }
            // Set the keyframe to the last keyframe
            currentKeyframe = Keyframes.Count - 1;
        }
        else
        {
            // If the animation is not looping,
            // then set the time to the beginning of the animation.
            currentPosition = TimeSpan.Zero;
        }
    }

    // Update the bone transforms based on the current time.
    UpdateBoneTransforms(timeSpan >= TimeSpan.Zero);
}
```

The `Update` method first updates the `currentPosition` with the elapsed time. It then check whether the animation has passed the end of the animation. If looping is used, then `currentPosition` is set back to the start of the animation; otherwise, it is set to the end `Length` of the animation. If the animation is run backwards, then check whether the `currentPosition` is less than `TimeSpan.Zero` and set the correct `currentPosition` before you finally call the `UpdateBoneTransforms` method.

Finally, implement the `UpdateBoneTransforms` method, which sets the current bone transforms to use based on the current keyframe time. Add the following method to the `AvatarCustomAnimation` class:

```
private void UpdateBoneTransforms(bool playingForward)
{
    if (playingForward)
    {
        while (currentKeyframe < Keyframes.Count)
        {
            // Get the current keyframe
            Keyframe keyframe = Keyframes[currentKeyframe];

            // Stop when we've read up to the current time.
            if (keyframe.Time > currentPosition)
                break;

            // Apply the current keyframe's transform to the bone array.
            avatarBoneTransforms[keyframe.BoneIndex] = keyframe.Transform;

            // Move the current keyframe forward.
            currentKeyframe++;
        }
    }
    else
    {
        while (currentKeyframe >= 0)
        {
            // Get the current keyframe
            Keyframe keyframe = Keyframes[currentKeyframe];

            // Stop when we've read back to the current time.
            if (keyframe.Time < currentPosition)
                break;

            // Apply the current keyframe's transform to the bone array.
            avatarBoneTransforms[keyframe.BoneIndex] = keyframe.Transform;
```

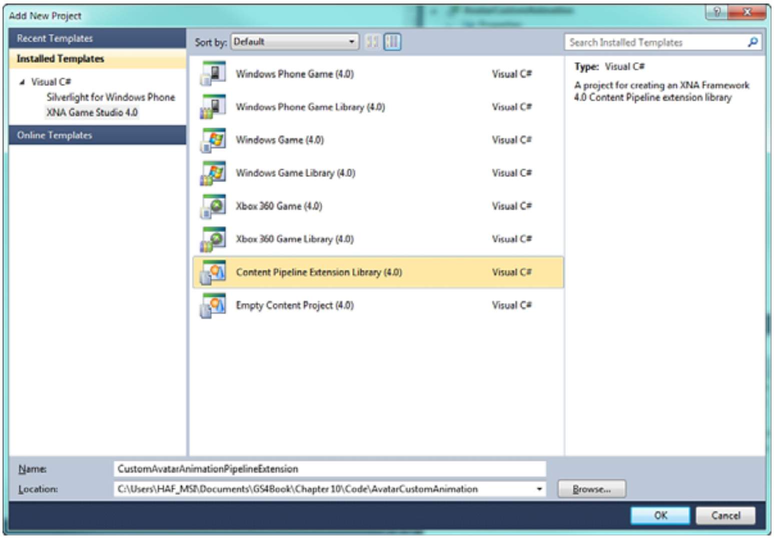
```
                // Move the current keyframe backwards.
                currentKeyframe–;
            }
        }
    }
```

`UpdateBoneTransforms` finds the current animation keyframes to use for the avatar bone transforms. The list of `Keyframes` is looped over starting at the `currentKeyframe` index. The loop stops when a keyframe's `Time` property is greater than the `currentPosition` in the animation or when there are no keyframes left.

Creating the Content Processor

Now that you have the types that make up the avatar's custom animation, create the custom content processor that converts the model's animation data into the format that you want at runtime. This is similar to the processor you wrote in Chapter 6, "Built-in Shader Effects" for the `SkinnedEffect`.

First, create a new pipeline extension project. Right-click the solution and select Add -> New Project. In the Add New Project dialog, select the Content Pipeline Extension Library project type and name it `CustomAvatarAnimationPipelineExtension`. Click OK as shown in Figure 10.15.


Add New Project
Recent Templates
Installed Templates
Visual C#
Silverlight for Windows Phone
XNA Game Studio 4.0
Online Templates
Search Installed Templates
Windows Phone Game (4.0) Visual C#
Windows Phone Game Library (4.0) Visual C#
Windows Game (4.0) Visual C#
Windows Game Library (4.0) Visual C#
Xbox 360 Game (4.0) Visual C#
Xbox 360 Game Library (4.0) Visual C#
Content Pipeline Extension Library (4.0) Visual C#
Empty Content Project (4.0) Visual C#
Browse...
OK
Cancel

Sort by: Default

Type: Visual C#

A project for creating an XNA Framework 4.0 Content Pipeline extension library

Name: CustomAvatarAnimationPipelineExtension

Location: C:\Users\HAF_MSI\Documents\GS4Book\Chapter 10\Code\AvatarCustomAnimation

Figure 10.15 Add New Project dialog with the Content Pipeline Extension Library project selected

Next, add two assemblies to the new content pipeline extension project. Right-click the References list and select Add Reference as shown in Figure 10.16.

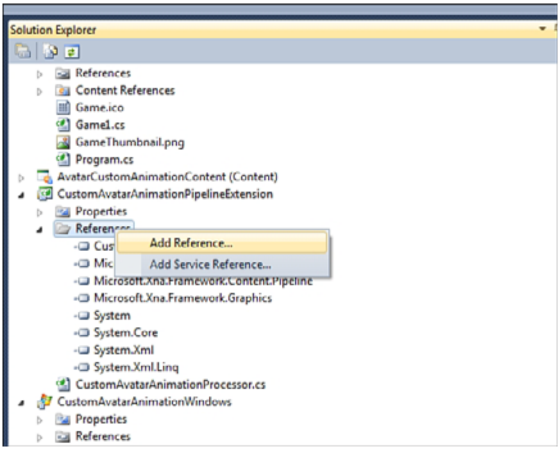

Solution Explorer
References
Content References
Game.ico
Game1.cs
GameThumbnail.png
Program.cs
AvatarCustomAnimationContent (Content)
CustomAvatarAnimationPipelineExtension
Properties
Referen
Cus
Mic
Add Reference...
Add Service Reference...
Microsoft.Xna.Framework.Content.Pipeline
Microsoft.Xna.Framework.Graphics
System
System.Core
System.Xml
System.Xml.Linq
CustomAvatarAnimationProcessor.cs
CustomAvatarAnimationWindows
Properties
References

Figure 10.16 Add Reference menu

Click the Projects tab and select the `CustomAvatarAnimationWindows` project that you previously created and click OK (see Figure 10.17).

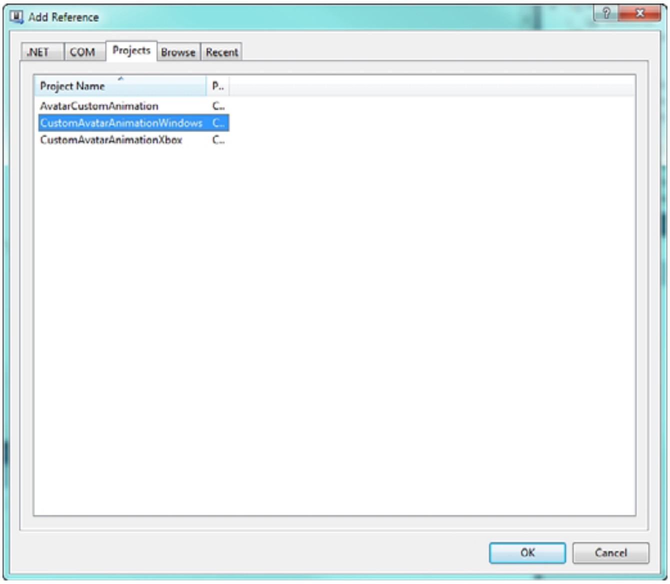
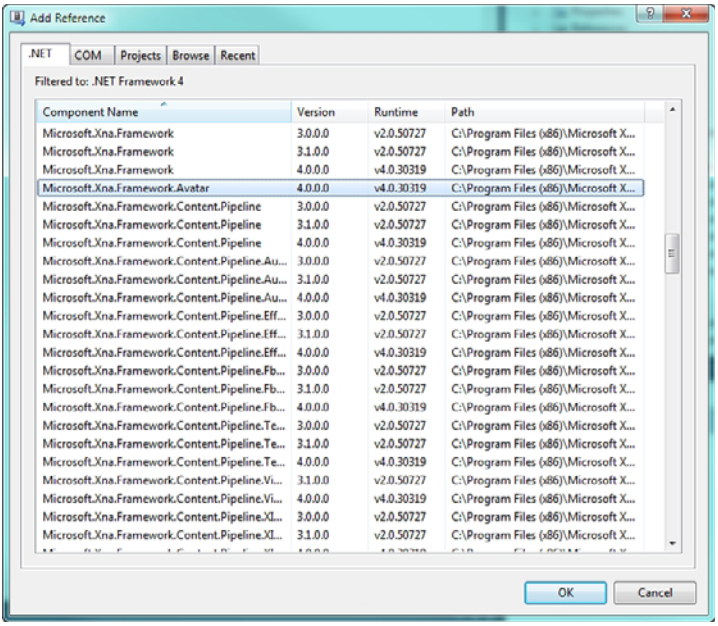

Add Reference
.NET
COM
Projects
Browse
Recent
Project Name
P...
AvatarCustomAnimation
C...
CustomAvatarAnimationWindows
C...
CustomAvatarAnimationXbox
C...
OK
Cancel

Figure 10.17 Adding project reference

Select the add reference menu again, but this time select the .NET tab. Scroll down, select the `Microsoft.Xna.Framework.Avatar` assembly, and click the OK button (see Figure 10.18).



Filtered to: .NET Framework 4

Component Name	Version	Runtime	Path
Microsoft.Xna.Framework	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Avatar	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Au...	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Au...	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Au...	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Eff...	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Eff...	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Eff...	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Fb...	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Fb...	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Fb...	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Te...	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Te...	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Te...	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Vi...	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.Vi...	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.XI...	3.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...
Microsoft.Xna.Framework.Content.Pipeline.XI...	3.1.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft X...

Figure 10.18 Adding reference to `Microsoft.Xna.Framwork.Avatar`

You need these two assemblies when you write the custom processor. The project file reference to the custom avatar types needs to be a Windows project because the processor will run on Windows during the build. Even if the content is built for the Xbox 360, the content project needs to reference Windows libraries.

Now you are ready to start writing the custom processor. First, add the following two namespaces to the list already defined in the processor file:

```
using Microsoft.Xna.Framework.GamerServices;
using CustomAvatarAnimation;
```

This enables you to use the `AvatarRenderer` and the custom animation types you created previously.

You now need to update the processor class definition and display name. Update the default processor class with the following class definition:

```
[ContentProcessor(DisplayName = "CustomAvatarAnimationProcessor")]
public class CustomAvatarAnimationProcessor :
                                ContentProcessor<NodeContent, CustomAvatar
➥AnimationData>
```

Use the `ContentProcessor` attribute to make this type as a content processor and to define the `DisplayName` that will be used within the Visual Studio content item property menu. The `CustomAvatarAnimationProcessor` class inherits from `ContentProcessor`, which is a generic abstract class that takes the input and output types as the generic parameters. `NodeContent` is the input type that is passed into the processor. You will create and return your `CustomAvatarAnimationData` type.

The processor contains a single field member to store a list of `Matrix` transforms that make up the bind pose of the avatar rig. Add the following member variable to the `CustomAvatarAnimationProcessor`:

```
// The bind pose of the avatar
List<Matrix> bindPose = new List<Matrix>();
```

The `ContentProcessor` defines a Process method, which converts the input type, for example, a `NodeContent` into the output type that is the `CustomAvatarAnimationData`. To override and provide an implementation for this method, add the following method to the `CustomAvatarAnimationProcessor` class:

```
public override CustomAvatarAnimationData Process(NodeContent input,
                                                  ContentProcessorContext context)
{
    // Find the skeleton of the model
    NodeContent skeleton = FindSkeleton(input);

    // We have to find the skeleton and it needs to have 1 animtion
    if (skeleton == null || skeleton.Animations.Count != 1)
    {
        throw new InvalidContentException("Invalid avatar animation.");
    }

    // Update the skeleton to what we expect at runtime
    CleanSkeleton(skeleton);

    // Flat list of the bones in the skeleton
    IList<NodeContent> bones = FlattenSkeleton(skeleton);

    // The number of bones should match what the AvatarRender expects
    if (bones.Count != AvatarRenderer.BoneCount)
    {
        throw new InvalidContentException("Invalid number of bones found.");
    }
    // Populate the bind pose list
    foreach (NodeContent bone in bones)
    {
        bindPose.Add(bone.Transform);
    }

    // Build up a table mapping bone names to indices
    Dictionary<string, int> boneNameMap = new Dictionary<string, int>();
    for (int i = 0; i < bones.Count; i++)
    {
        string boneName = bones[i].Name;
        if (!string.IsNullOrEmpty(boneName))
        {
            boneNameMap.Add(boneName, i);
        }
    }

    CustomAvatarAnimationData avatarCustomAnimationData = null;
    foreach (KeyValuePair<string, AnimationContent> animation in skeleton.Animations)
    {
        // Animation duration needs to be greater than 0 length
        if (animation.Value.Duration <= TimeSpan.Zero)
        {
            throw new InvalidContentException("Animation has a zero duration.");
        }

        // Build a list of the avatar keyframes in the animation
        List<Keyframe> animationKeyFrames = ProcessAnimation(animation.Value,
                                                             boneNameMap);

        // Check for an invalid keyframes list
        if (animationKeyFrames.Count <= 0)
        {
            throw new InvalidContentException("Animation has no keyframes.");
        }

        // Create the custom-animation object
        avatarCustomAnimationData = new CustomAvatarAnimationData(animation.Key,
                                                         animation.Value.Duration,
                                                         animationKeyFrames);
    }

    return avatarCustomAnimationData;
}
```

This is a long method, so let's take a look at it piece by piece. The first thing you do is call the `FindSkeleton` method to location where in the input node tree the skeleton exists. The skeleton is where you find the animation data that you need to use. If you are unable to locate the skeleton data, throw an exception because there is nothing you can do with the content file.

Next, you pass the skeleton into a method called `CleanSkeleton`. The skeleton is exported from the avatar animation rig that contains bones that are not used at runtime, so remove them. Also, clean up some of the naming that might be used in the rig. If you remove a bone from the skeleton, make sure you don't process the keyframes from the bone later in the processor.

Now, you have a clean skeleton hierarchy, but you really want a flattened list of bones that match what is expected by the `AvatarRenderer`. This list of bones is sorted by the depth of the bone in the hierarchy and within a level they are sorted by the name of the bone. You call the `FlattenSkeleton` method to convert the `NodeContent` skeleton hierarchy into the sorted flat list you want. Check that the number of bones returned from the `FlattenSkeleton` method equals the number of bones in the `AvatarRenderer` using the `BoneCount` constant value.

Now that you have the real list of bones, save their transform value. The transform set on each of the bones is called the bind pose. This is the starting location of the animation rig before the animator changes the rig to create the animations. The animation keyframe transforms are relative to this starting position called the bind pose. You loop over all of the bones and add their transforms to the `bindPose` list.

When you process the animation keyframes, you will have the `string` name of the bone they transform. The custom avatar animation needs the index value of the bone. To be able to find the bone index, create a `Dictionary` of `string` and `int` values that store the name of the bone and the index of the bone. This enables you to quickly look up the index of a bone for a given `string` name. To populate the `Dictionary`, loop over the flat list of bones and add the name of the bone and the index value to the `Dictionary`.

The final state of the `Process` method is to convert the animation data into the format, which is a list of keyframes that you will use to construct the new `CustomAvatarAnimationData` object.

Loop over the animations that are attached to the skeleton. The `ProcessAnimation` method is called passing in the `AnimationContent` and the `boneNameMap` Dictionary you created. A list of `Keyframe` values returns that is then used to construct the new `CustomAvatarAnimationData` before returning it from the `Process` method.

The `Process` method called into a number of helper methods that you now need to create. The first is the `FindSkeleton` method. Add the following method to the `CustomAvatarAnimationProcessor` class:

```
private NodeContent FindSkeleton(NodeContent input)
{
    // This is the node we are looking for
    if (input.Name.Contains("BASE__Skeleton"))
    {
        return input;
    }

    // Recursively check all children until we find the root of the skeleton
    foreach (NodeContent child in input.Children)
    {
        NodeContent skeleton = FindSkeleton(child);

        if (skeleton != null)
            return skeleton;
    }

    return null;
}
```

The root of the skeleton in the avatar rig is called BASE__Skeleton. Check whether the current `NodeContent` has the same name. If you have not found the root of the skeleton, loop over all of the children of the current node and recursively call the `FindSkeleton` method for each of the children.

To flatten the skeleton hierarchy, implement the `FlattenSkeleton` method. Add the following method to the processor:

```
// Flatten the skeleton into a list ordered by level depth
static IList<NodeContent> FlattenSkeleton(NodeContent skeleton)
{
    // Return list of bones we find in the skeleton
    List<NodeContent> bones = new List<NodeContent>();

    // Skeleton bones in the current level
    List<NodeContent> currentLevelBones = new List<NodeContent>();

    // Start with the root node
    currentLevelBones.Add(skeleton);

    while (currentLevelBones.Count > 0)
    {
        List<NodeContent> nextLevelBones = new List<NodeContent>();

        // Avatar bones are sorted by name in each level
        IEnumerable<NodeContent> sortedBones = from item in currentLevelBones
                                                orderby item.Name
                                                select item;

        // Add the sorted list to our list
        foreach (NodeContent bone in sortedBones)
        {
            bones.Add(bone);
            // Add all of the children for the next level
            foreach (NodeContent child in bone.Children)
            {
                nextLevelBones.Add(child);
            }
        }

        currentLevelBones = nextLevelBones;
    }

    return bones;
}
```

To flatten the skeleton, create a list of `NodeContent` instances called `bones`. This is the final list that you return from the method. You also need a list that stores the `NodeContent` instances that are at the same depth in the skeleton hierarchy and have the same level. The final list needs to be sorted by level and then by name within the level.

Create a loop that continues until the current level contains no bones. The first level is the root so it contains only the single item. As you process each level, sort the bones in the level, and then add the children of the current level into a list for the next level. Continue this looping until there are no children left to process. The resulting list is correctly sorted for use with the `AvatarRenderer`.

The `CleanSkeleton` method is used to remove bones that are not needed at runtime and to fix the names of the bones. Add the following method to the `CustomAvatarAnimationProcessor` class:

```
// Removes bones not used in the AvatarRenderer at runtime
// and fixes the names of some bones
static void CleanSkeleton(NodeContent bone)
{
    // Remove unwated text from the bone name
    bone.Name = bone.Name.Replace("__Skeleton", "");

    // Process all of the children
    for (int i = 0; i < bone.Children.Count; ++i)
    {
        NodeContent child = bone.Children[i];
        if (child.Name.Contains("_END"))
        {
            bone.Children.Remove(child);
            —i;
        }
        else
        {
            CleanSkeleton(child);
        }
    }
}
```

Loop over each of the children bones looking for any bone that contains _END. You don't need these bones, so remove them from the hierarchy.

The final two methods are responsible for converting the animation data from the content pipeline format `AvimationContent` into a list of `Keyframe` objects that you use within the custom animation. Add the following two methods to your `CustomAvatarAnimationProcessor` class:

```
// Convert animation from content pipeline format to a list of keyframes
List<Keyframe> ProcessAnimation(AnimationContent animation,
                                Dictionary<string, int> boneMap)
{
    // Return keyframe list
    List<Keyframe> keyframes = new List<Keyframe>();

    foreach (KeyValuePair<string, AnimationChannel> channel in animation.Channels)
    {
        // Don't process the end bone channel. We have removed these from the
➥skeleton
        if (channel.Key.Contains("_END"))
            continue;

        // Find which bone this channel has keyframes for
        int boneIndex;
        if (!boneMap.TryGetValue(channel.Key.Replace("__Skeleton", ""), out
➥boneIndex))
        {
            throw new InvalidContentException(string.Format(
                "Found animation for bone '{0}', " +
                "which is not part of the skeleton.", channel.Key));
        }

        // Craete the keyframes for the channel
        foreach (AnimationKeyframe keyframe in channel.Value)
        {
            keyframes.Add(new Keyframe(boneIndex,
                                       keyframe.Time,
                                       CreateKeyframeMatrix(keyframe,
boneIndex)));
        }
    }

    // Sort the final list of keyframes by time
    keyframes.Sort((frame1, frame2) => frame1.Time.CompareTo(frame2.Time));

    return keyframes;
}
```

```
// Convert animation keyframes info the format used by the AvatarRenderer
Matrix CreateKeyframeMatrix(AnimationKeyframe keyframe, int boneIndex)
{
    Matrix keyframeMatrix;

    // The root node is transformed by the root of the bind pose
    // We need to make the keyframe relative to the root
    if (boneIndex == 0)
    {
        // If you are using an older verion of the FBX exporter the root
        // of the bind pose my be translated incorrectly
        // If your model appears to be floating use the following translation
        //Vector3 bindPoseTranslation = new Vector3(0.000f, 75.5199f, -0.8664f);
        Vector3 bindPoseTranslation = Vector3.Zero;

        Matrix inverseBindPose = bindPose[boneIndex];
        inverseBindPose.Translation -= bindPoseTranslation;
        inverseBindPose = Matrix.Invert(inverseBindPose);

        Matrix keyTransfrom = keyframe.Transform;
        keyframeMatrix = (keyTransfrom * inverseBindPose);
        keyframeMatrix.Translation -= bindPoseTranslation;

        // Scale from cm to meters
        keyframeMatrix.Translation *= 0.01f;
    }
    else
    {
        keyframeMatrix = keyframe.Transform;
        // Remove translation from anything by the root
        keyframeMatrix.Translation = Vector3.Zero;
    }

    return keyframeMatrix;
}
```

The `ProcessAnimation` starts by creating a new list of `Keyframe` instances that you use to store the newly created `Keyfame` values. Loop each of the `Channels` in the `AnimationContent`. An `AnimationChannel` contains all of the `keyframes` for a specific bone in the skeleton over the course of the animation.

As you loop over the animation channels, don't process any of the channels for the bones that include _END in their name. These were the bones that you removed when you flattened the skeleton, and you don't need them in the animation.

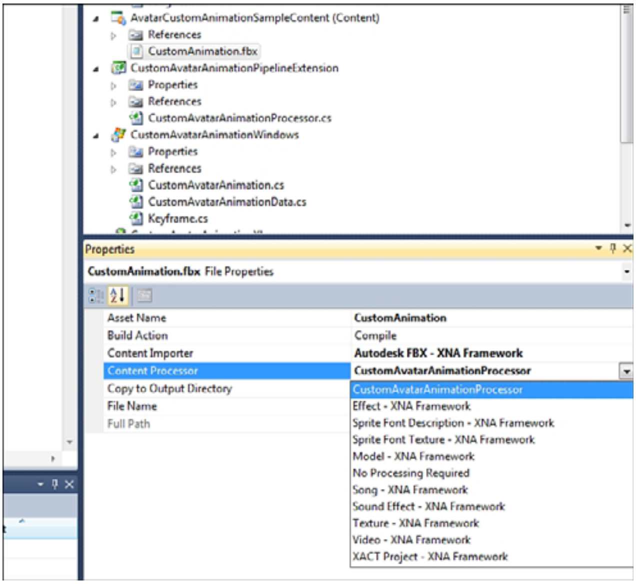
Store the bone index for each keyframe in the animation. To get the bone index, perform a lookup into the `boneMap Dictionary`. After you have the bone index, create the new `Kayframe` object using the index, the keyframe's `Time`, and a transform matrix that you create using the `CreateKeyframeMatrix` method.

Adding the Custom Animation to Your Game

Now that the processor is done, add a reference to the `CustomAvatarAnimationPipelineExtension` project to the content project. Expand the `AvatarCustomAnimationSampleContent` project. Right-click the References list and select Add Reference. Select the Projects tab and select the `CustomAvatarAnimationPipelineExtension` project. Next, add the animation you built to the content project. Right-click the `AvatarCustomAnimationSampleConent` project and select Add -> Existing Item. Locate the animation fbx file that you created in your 3D content creation package and click the Add button.

With the new content item added to the content project, we n change the processor to use when building the content item. Click the animation fbx file in the content project and locate the properties panel. Under the properties panel, in Content Processor, click the drop-down and select `CustomAvatarAnimationProcessor` as shown in Figure 10.19.

The sample project, `AvatarCustomAnimationSample`, is an Xbox 360 project, but it should use the types you created previously in the `CustomAvatarAnimationWindows` project. To use these types, build them as a library for the Xbox 360. Fortunately, this is a common scenario and an easy way to create the Xbox 360 version of the library. Right-click the `CustomAvatarAnimationWindows` project and select the Create Copy of Project for Xbox 360 menu option as shown in Figure 10.20.


AvatarCustomAnimationSampleContent (Content)
References
CustomAnimation.fbx
CustomAvatarAnimationPipelineExtension
Properties
References
CustomAvatarAnimationProcessor.cs
CustomAvatarAnimationWindows
Properties
References
CustomAvatarAnimation.cs
CustomAvatarAnimationData.cs
Keyframe.cs
Properties
CustomAnimation.fbx File Properties
CustomAvatarAnimationProcessor
Effect - XNA Framework
Sprite Font Description - XNA Framework
Sprite Font Texture - XNA Framework
Model - XNA Framework
No Processing Required
Song - XNA Framework
Sound Effect - XNA Framework
Texture - XNA Framework
Video - XNA Framework
XACT Project - XNA Framework

Asset Name	CustomAnimation
Build Action	Compile
Content Importer	Autodesk FBX - XNA Framework
Content Processor	CustomAvatarAnimationProcessor
Copy to Output Directory	
File Name	
Full Path	

Figure 10.19 Selecting the `CustomAvatarAnimationProcessor` content processor

After the new project is created, rename the project `CustomAvatarAnimationXbox`. Your `AvatarCustomAnimationSample` project now needs a reference to the new project, so expand the `AvatarCustomAnimationSample` project and right-click the Reference list. Select the Add Reference menu item. Click the Projects tab and select the `CustomAvatarAnimationXbox` project.

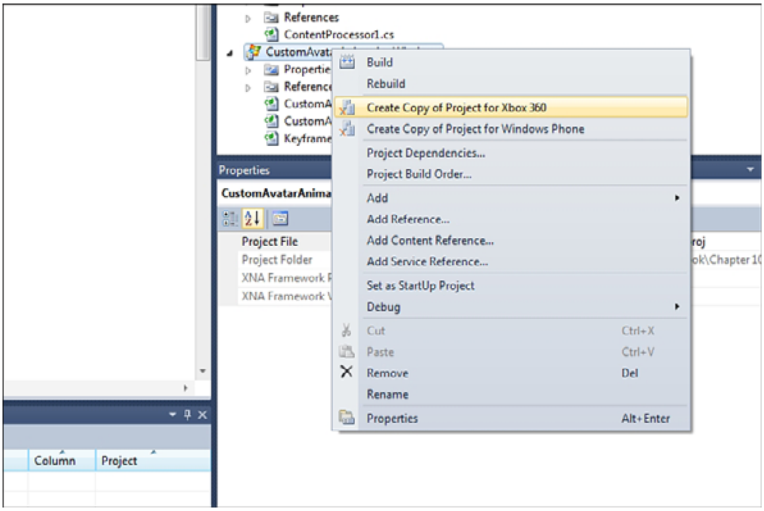

References
ContentProcessor1.cs
CustomAvat
Propertie
Referenc
CustomA
CustomA
Keyframe
Properties
CustomAvatarAnima
Project File
Project Folder
XNA Framework
XNA Framework
Column
Project
Build
Rebuild
Create Copy of Project for Xbox 360
Create Copy of Project for Windows Phone
Project Dependencies...
Project Build Order...
Add
Add Reference...
Add Content Reference...
Add Service Reference...
Set as StartUp Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Properties Alt+Enter

Figure 10.20 Creating copy of a project for Xbox 360

Updating Your Game to Use the Custom Animation

You can use the random avatar example that you created at the beginning of this chapter. You need to make only a few minor updates to use the new custom animation type.

First, add the namespace you used for the custom animation types by adding the following line to your game class file:

```
using CustomAvatarAnimation;
```

Next, you need a member variable to store the custom animation. Add the following member variable to your `Game` class:

```
AvatarCustomAnimation customAvatarAnimation;
```

To load the `AvatarCustomAnimation`, add the following lines to your `LoadContent` method:

```
// Load the custom animation
CustomAvatarAnimationData animationData =
Content.Load<CustomAvatarAnimationData>("CustomAnimation");
customAvatarAnimation = new AvatarCustomAnimation(animationData);
```

The fbx file is converted into a `CustomAvatarAnimationData` using the custom content pipeline processor. Then, use the `ContentManager` to load the file. You can then create a `AvatarCustomAnimation` instance using the `CustomAvatarAnimationData`.

Now, you can use the `AvatarCustomAnimation` just like you use an `AvatarAnimation`. In the game's `Update` method, call the animations `Update` method. Add the following to your game's `Update` method:

```
customAvatarAnimation.Update(gameTime.ElapsedGameTime, true);
```

The final step is to use the animation with the `AvatarRenderer Draw` method. Add the following to your game's `Draw` method.

```
avatarRenderer.Draw(customAvatarAnimation);
```

Running the sample now shows a random avatar playing back your custom animation. Figure 10.21 shows the avatar playing the custom animation created for this sample.

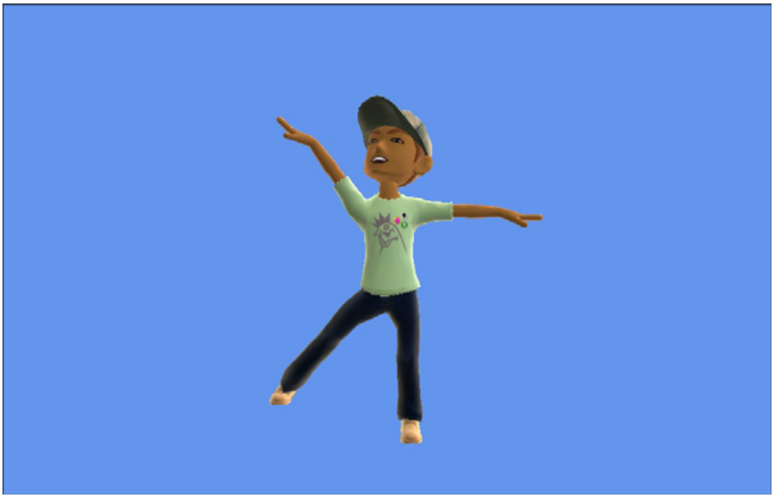


Figure 10.21 Custom animation playing on the avatar

Summary

We have covered a large amount of information about the avatar library provided by XNA Game Studio 4.0 and built several samples. We discussed the basics of using the `AvatarDescription`, `AvatarAnimation`, and `AvatarRenderer` APIs to more advanced samples such as animation blending and how to load and play back custom animations.

You now have the power to utilize avatars in your own Xbox Live Indie game, so create an awesome new avatar gaming experience.

This page intentionally left blank

11

Understanding Performance

There is an old adage in which there are two steps you need to follow to optimize your performance. Step one is "Don't do it." Step two, which for experts is "'Don't do it yet." This is good advice, so let's make a goal in this chapter to get you on your way to step two. In this chapter, you' learn:

- General performance guidelines
- Graphics performance
- Measuring performance

One of the biggest concerns people seem to have (even to this day) is that managed code cannot perform well enough. This was somewhat understandable years ago when managed code was new and everyone wasn't sure how to write code for it, but the managed runtime has been out for years now. The fourth version came out with Visual Studio 2010, and it's been more or less proven that you can write high-performing managed applications.

General Performance

Some people mistake that saying at the beginning of the chapter to mean that you shouldn't worry about performance, which is just not accurate. The intent is to warn people about the potential problems that can arise from attempting to prematurely optimize your code. Optimizing code that is already "fast enough" just makes the code harder to read and maintain, or worse yet, it can introduce new bugs and regressions into the code. Performance is important, but just like anything else, you shouldn't try to fix it, unless you know what you're off to fix and why.

One of the important aspects of getting great performance out of your applications (and games) is an understanding of what various portions of your code cost. For example, do you know what the following code snippet might cost?

```
MediaPlayer.Play(Song.FromUri("mysong.wma", new Uri("http://www.someplace.com")));
```

If you guessed quite a lot, you'd be correct! Although it is a single line of code, there are a whole lot of things going on here. There are two orphaned objects (the `Uri` and the `Song`) that will eventually be collected by the garbage collector. There is `FromUri` method that (with the code used here) attempts to connect to the Internet to try to download music. What if you were on a phone with no connectivity? What if your latency was so high, it took 5 seconds to even start the download? Does the method download the entire file before attempting to play or does it try to stream? Does the method return instantly or block until sometime in the future when it has a valid song object? You should have a basic idea of the performance (or cost) of a method before calling it.

Cost can mean a number of different things. Perhaps a method allocates an extraordinary amount of memory; this would certainly add to its cost, potentially in a variety of ways. For example, it can cause the garbage collector to collect more frequently or it can push data out of the cache. A method can simply be computationally expensive, because it's a complex physics engine handling millions of objects. You can have a method that is fast in most cases, but sometimes it needs to wait for data and blocks execution until it has it.

One easy-to-hit performance problem in early versions of the managed runtimes was what is called boxing. In .NET, anything can be used as an object, which is a reference type. Before we move on to boxing, what exactly is a reference type?

At the most basic level, a reference type is a type that you do not access the value of directly, but through a reference (hence, the name). This means that the object is allocated on the heap, and the garbage collector collects it later when it no longer has any references. Because reference types are referred to by reference, assigning two variables to the same reference means each variable has the same data. For example, imagine a class with a method `SetInteger` that sets an integer and `GetInteger` that returns the value of that integer, which is shown in this code snippet:

```
MyClass firstRef = new MyClass();
MyClass secondRef = firstRef;
secondRef.SetInteger(5);
firstRef.SetInteger(3);
int x = secondRef.GetInteger();
```

This produces a value of 3 for the variable x. This is because each variable refers to the same reference, so modifying one naturally modifies the other. If a type isn't a reference type, then it is considered a value type. Most integer primitives (and structs) are value types, and they are quite different than reference types because assigning one value type to another does a full copy rather than make the variables share the same reference. For example, in the previous example, if you were using integers directly (which are value types) rather than reference types, you would get a different result:

```
int firstInt = 5;
int secondInt = firstInt;
secondInt = 3;
firstInt = 7;
int x = secondInt;
int y = firstInt;
```

At the end of this snippet `x` is 3, and `y` is 7. Unlike reference types, changing one doesn't change the other. A common misconception of value types is that they are always on the stack. If they are declared as a local variable, then this is normally true, but they are not always on the stack. For example, an array of value types is a reference type, and each individual value type is on the heap. Similarly, if a value type is a member of a class, it is stored on the heap, too.

Value types that are local variables can sometimes be performance wins, though, as they are allocated on the stack, and hence, they are not going to be garbage collected later. However, they can also be a performance hit if you don't know what you're doing with them. For example, if you have a large value type (such as the `Matrix` structure), and you declare six of them as local variables so they are on the stack, but then pass all six into a method, this may be a performance hit even though they are on the stack! Because assigning a value type creates a deep copy, you allocate 12 of the structures in total (six for your original local variables and six for the method parameters), plus (and more importantly) you copy all the data for each of those structures into the parameters (the `Matrix` structure is 64 bytes, so that is copying 384 bytes just to call that method). Again, this is more affirmation that you should have a rough idea of the cost of your methods.

So, now we come back to boxing. Although this class of performance behavior is much less prominent than before, it can still be hit. The act of boxing is essentially taking a value type on the stack, and using it as an object type (which is a reference type). Doing this allocates a new reference type and stores the data of the value type inside it. The potential pitfall here is that all of those new objects need to be collected, and this causes the garbage collector to kick in sooner. With the advent of generics, the majority of common cases where people accidently box something are gone, but not all of them.

Who Takes Out the Garbage?

By far, the most common cause of unexpected performance problems is related to the garbage collector. For the most part, modern day garbage collectors are good at what they do, so before we get into potential performance issues, let's first take a look at what a garbage collector actually does.

In the .NET runtime, objects are created on the managed heap, which is in essence a large block of memory. The runtime keeps track of this memory and remembers where the start of the memory block is. When you allocate new objects, it is almost free because all the runtime needs to do is move its pointer at the top of the heap down, as seen in Figure 11.1.

As you allocate more and more objects, some go out of scope, some stay in scope, and that allocation pointer just keeps moving down the block of memory. Eventually, the system decides to check to see if you are still using all the memory you allocated. It does that by starting at the beginning of the memory block and examining all of the objects that have been created. If it finds an object that is orphaned, it marks that object, and then it continues on.

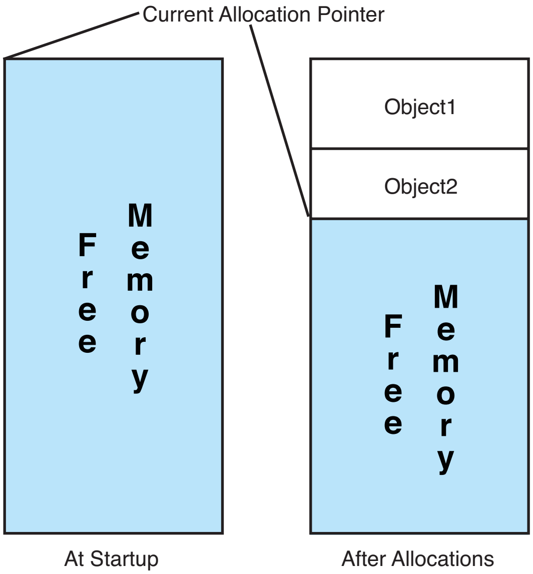

Current Allocation Pointer
Free Memory
At Startup
Object1
Object2
Free Memory
After Allocations

Figure 11.1 Basic memory allocation in the managed runtime

Note

All of the points made here about the garbage collector are at a high level. Many resources are available to learn more about the garbage collector if you want more in depth detail.

An object is orphaned when it has no outstanding references to it anywhere in the application. For example, in this code snippet, the object `o` is orphaned as soon as the method returns and is available to be collected:

```
public void SomeMethod()
{
  object o = new object();
}
```

However, in this snippet, the object `o` is not orphaned when the method ends because the class itself still holds a reference to it:

```
public class SomeClass
{
  object o;
  public void SomeMethod()
  {
     o = new object();
  }
}
```

If the class becomes orphaned, then the object contained in it does, too (provided it doesn't have other outstanding references).

After the garbage collector has gone through the entire memory block finding orphaned objects, it goes and compacts the memory block. This means it goes through that memory block and moves things around until it has a single, big contiguous memory block again, destroying objects that have no references.

The Windows desktop runtime is actually even a little smarter than that. It has what is called a generational garbage collector. When an object is first created, it assumes it will be short lived, and it will be created in what is called generation zero, for the short lived objects. When a certain threshold is hit, the runtime decides to do a collection, and it looks only through the generation zero objects, marks the ones that are orphaned, and then collects them. However, anything that is in generation zero that is not orphaned gets promoted to the next generation (generation one). If another threshold is met, a generation one collection occurs, and things not orphaned get promoted to generation two, which is the end of the line for the objects. A generation two collection is the one that walks the entire memory block.

As you can imagine, examining every object you create, and then compacting the memory after a collection is time-consuming, particularly if you have a large number of objects. Because the generation two collection is the one that looks at all objects, it is the worst performing collection of the group.

However, the compact framework runtime that runs on the Xbox and Windows Phone 7 does not have a generational garbage collector. Every collection on those platforms looks at every object. If the garbage collector runs all the time, your game sees noticeable slowdowns.

> **Note**
>
> The garbage collector has a set of heuristics that determine when it should start a collection, but they are not configurable, and you cannot predict when it will happen.

I'm sure the question then is, "Well, how do I avoid that?" The answer is easy; don't create garbage! Now, obviously you can't run your entire game without creating objects. You just need to make sure that you don't create objects all the time during performance-critical sections of your game (such as during game play). For example, if you create a new state object every frame to set the device state, you will have collections during game play.

Normally though, it's the unexpected allocations that cause the collections, more proof that you want to have an idea of what methods cost. As an example, did you know that using `foreach` on a collection can potentially create garbage? This is what `foreach` breaks down into:

```
IEnumerator e = collection.GetEnumerator();
while (e.MoveNext())
{
    var obj = e.Current;
    // Your code
}
```

That first line creates an object. If that object is not a value type, that object is created on the heap and produces garbage. Most collection types in the runtimes do not create garbage during a `foreach` loop, but the fact that it can is often a surprise to people.

Later in this chapter we discuss how to measure your garbage.

Multithreading

Another area where you can see an improvement in performance (particularly on Xbox) is in multithreading. Now, this doesn't mean you should add multithreading everywhere because poor use of multithreading can actually make performance worse.

If you have portions of your code that are computationally expensive and don't require them to be done at a certain point in time (for example, you aren't waiting on the results), using another thread can be a big win for you. Creating a thread and starting one can take a bit of time, though, so if think you need multiple threads, you should create them all at startup to not have to pay that cost at runtime.

Another thing to keep in mind when using multiple threads on Xbox is what core they run on. The Xbox has three physical cores each with two hardware threads, which means it can run six threads simultaneously. However, unlike on Windows, you must explicitly tell it which thread you want to run on. You do this with the `SetProcessorAffinity` method as you see in the following snippet:

```
Thread worker = new Thread(new ThreadStart(() =>
    {
        Thread.CurrentThread.SetProcessorAffinity(4);
        DoComplexCalculations();
    }));
worker.Start();
```

> **Note**
>
> You need to set the processor affinity as the first line in your thread method; you cannot set it before the thread has started nor can you call it on any thread other than the one on which it executes.

Although the Xbox has a total of six hardware threads you can use, two of them are reserved by the XNA runtime (thread zero and thread two). You should use the other four hardware threads to spread your calculations across each of the hardware threads.

One thing you want to avoid in a multithreaded game is waiting on one of your worker threads to continue the game. If your game is paused to wait for another thread, you potentially don't gain any performance.

Let's imagine a scenario that has characteristics such as this. Your `Update` method tells a worker thread to perform a task that takes 1ms, while the rest of the code in the `Update` method also takes 1ms to complete. However, it needs the rest of the data from the worker thread, so it needs to wait for that thread to finish, which after the overhead, the thread can take more than the 1ms you attempted to save, and you would take less time by doing all the work in your `Update` method.

Conversely, imagine your `Update` method takes 7ms to complete, and the worker thread takes 8ms to complete. Sure, your `Update` method would end up waiting a millisecond or two waiting on the data, but that's much faster than the 15ms it would have had to wait if it was all done in the `Update` call.

Entire books are written about writing well-behaving multithreaded code, and this small section doesn't even begin to touch the surface of the things you need to watch out for (for example, the `lock` keyword in C#). Hopefully, this gives you some ideas about how multiple threads can help your performance without giving you the false sense that is the panacea to make your performance troubles go away.

Graphics Performance

For some reason, when people discuss performance in games, a lot of times, they talk about graphics performance specifically. It's rare they speak of physics performance or audio performance—it's almost always graphics. This is probably because many people don't understand graphics performance or what it means.

One of the biggest conceptual problems people have when attempting to understand graphics performance is that it doesn't behave similarly to other parts of the system. In a normal application, you tell the computer what you want it to do, and it generally does exactly what you ask it to do in the order you ask it to do it.

Modern graphics hardware doesn't behave anything like that. It has a bunch of stuff to do and it says, "Hey, I'm just going to do it all at once." Top of the line graphics cards today can process thousands of operations simultaneously.

On top of that, people can get confused because they measure performance and see that their `Draw` calls are fast, but then there is a slowdown in the `Present` method that they didn't even call! This is because anytime a graphics operation is called, the computer doesn't actually tell the graphics hardware until later (normally, the `Present` operation); it just remembers what you asked it to do, and then it batches it up and sends it all to the graphics hardware at once.

Because the graphics hardware and the computer run at the same time, you might have realized that this means the graphics hardware will render your scene while your next frame's `Update` is executed, as you see in Figure 11.2. This is the basis of knowledge you need to understand to make good decisions about graphics performance.

In a perfect world, this is how all games would run. As soon as the graphics hardware is done drawing the scene, the CPU gives it the next set of instructions to render. Neither piece of hardware waits, on the other, and they run perfectly in sync. This rarely happens, though; normally one waits for the other.

Let's look at hypothetical numbers for a game. The work the CPU does takes 16ms, and the graphics hardware renders the scene in 3ms. So now, when the CPU has told the graphics hardware to start rendering and restarts its loop, the graphics hardware is done rendering and waits for its next set of instructions while the CPU still has 13ms of work left to do before it gives the graphics hardware anything new to do.

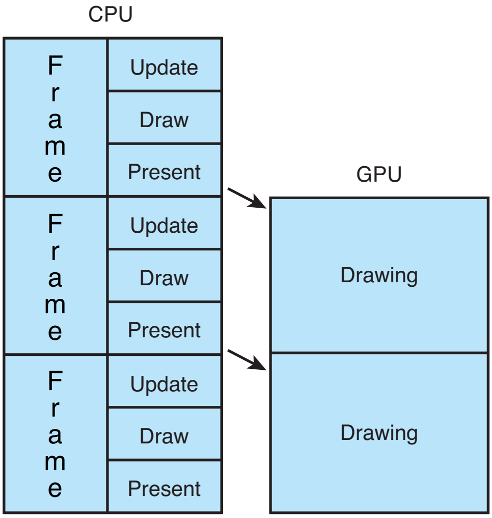


Figure 11.2 The perfect balance of CPU and GPU

This is called being *CPU bound*, and it is important to know the distinction between it and"GPU bound (which is discussed later in this chapter). If your game is CPU bound and you spend a bunch of time optimizing your graphics performance, you won't see a single bit of improvement, because the graphics hardware is already sitting around idle! In reality, you can actually add more graphics features here for free (if they were completely on the GPU), or you move some of your code from the CPU to the GPU.

Conversely, if these numbers are swapped and your CPU takes 3ms to do its work while the graphics hardware takes 16ms to render the scene, you run into the situation where the CPU is ready to give the next set of instructions to the graphics hardware only for the graphics hardware to complain, "I'm not done with the last section yet, hold on a second." Now, the CPU sits around waiting for the GPU, and this is called being GPU bound. In this scenario, optimizing your CPU code to run faster has no impact on performance because the CPU is already sitting around waiting as it is.

Knowing if your game is CPU bound or GPU bound can go a long way in determining how to address performance issues. The obvious question here is, "How do I detect which one I am?" Unfortunately, it's not as easy an answer as you migth think.

Sometimes you might notice that the `Present` call takes an awfully long time, causing your game to slow down. Then you think, "If `Present` takes so long, that must mean the graphics hardware is still busy, so I must be GPU bound!" This might be the case, but it can also be the case that you simply misinterpreted what happened in `Present`!

By default, an XNA game runs in fixed time step mode and runs with `SynchronizeVerticalRetrace` set to true. When this property is true, it tells the graphics hardware that it should render to the screen only when the monitor refreshes. A typical monitor has a refresh rate of 60hz, so if the monitor has that refresh rate, it refreshes

60 times per second (or every 16.6667ms). If rendering code takes 2ms to complete, the driver can still wait another 14.66667ms for the monitor refresh before completing the draw. However, if that property is set to false, the graphics hardware attempts to draw as fast as it can.

> **Note**
>
> Naturally, if your monitor runs at a different refresh rate from 60Hz, then the previous numbers would be different.

Of course, if you turn this off, you run into the opposite problem. Now, your graphics hardware runs as fast as it can, but because the system runs on a fixed time step (which is set to run at the default target speed for the platform: 60 frames per second on Xbox and Windows and 30 on Windows Phone 7), if you measured, you would appear to be CPU bound. This is because the game sits around and does not continue the loop until your next scheduled `Update` call!

So, to get a true measure of your performance, you need to run with `SynchronizeVerticalRetrace` and `IsFixedTimeStep` set to false. We recommend you create a new configuration called Profile to do your performance measurements and testing.

> **Note**
>
> Sometimes, any graphics call can cause a huge spike. This happens when the internal buffer being used to store graphics commands gets full. At this time, they're all sent to the hardware for processing. If you see this happen, you probably should think about doing fewer operations per frame by batching them up.

With these two items taken care of, let's take a look at measuring performance in your game.

Measuring Performance

One of the team members of the XNA Framework developed a helper library that he uses in all of his examples to help him easily measure performance of particular things. He was kind enough to give his permission to include this library in this book, which you can find with the downloadable examples for the book.

It wouldn't be possible to show an example game that had every possible performance penalty in it, but let's create a new project to see this library and get a few measurements. After you've created the project, right-click the solution, and choose Add->Existing Project, and then select the `TimeRulerLibrary` from the downloadable examples.

There's one thing you need to do before using this library. Because the library renders text, it needs to load a sprite font. It also expects to find the sprite font in a particular place, so right-click your `Content` project and choose Add->New Folder. Name it `Debug`. Then, right-click the newly created folder and choose Add->New Item, selecting a sprite font and naming it `DebugFont`. You need to change a few of the default options that sprite fonts have. Replace the `FontName`, `Size`, and `CharacterRegions` nodes with the following:

```
<FontName>Consolas</FontName>
<Size>10</Size>
<CharacterRegions>
  <CharacterRegion>
    <Start>&#32;</Start>
    <End>&#126;</End>
  </CharacterRegion>
  <CharacterRegion>
    <Start>&#x2582;</Start>
    <End>&#x2582;</End>
  </CharacterRegion>
</CharacterRegions>
```

With that, you're ready to use this helper library. The basic goal of the library is to give you detailed information about actions that occur in your game. It renders a frame counter, and you can also tell it to render little bars that measure how long things take. To make it easy to use the library, let's create a couple of helper objects. Add a new code file to your game project called `PerfHelper.cs`. Add a few objects to make it easier to use the library.

First, add a static class that will be the bulk of your interaction with the helper library:

```
public static class PerformanceHelper
{
    private static TimeRuler _currentRuler;
    private static DebugManager _debugManager;
    private static DebugCommandUI _debugCommandUI;
    private static FpsCounter _fpsCounter;
    private static bool _firstFrame = true;
}
```

The variables here are the common objects you will use in the library. The `TimeRuler` object is the one you will get the most use out of, as it is the one that renders the bars (or rulers) you'll use to measure the time it takes to run sections of code (hence, `TimeRuler`). The next few objects are used to help hold internal state; and the last is the component that displays your FPS counter so you can see how fast your game runs at a high level. Lastly, there are a couple things to do on the first frame, so you can store that, too.

Each of these objects is a component you can add to your game and forget about. You'll want to add this method to help do so easily:

```
public static void InitializeWithGame(Game g)
{
    // Initialize debug manager and add it to components.
    _debugManager = new DebugManager(g);
    g.Components.Add(_debugManager);
    // Initialize debug command UI and add it to compoents.
    _debugCommandUI = new DebugCommandUI(g);
    // Change DrawOrder for render debug command UI on top of other
compoents.
    _debugCommandUI.DrawOrder = int.MaxValue;
    g.Components.Add(_debugCommandUI);
    // Initialize FPS counter and add it to compoentns.
    _fpsCounter = new FpsCounter(g);
    g.Components.Add(_fpsCounter);
    // Initialize TimeRuler and add it to compoentns.
    _currentRuler = new TimeRuler(g);
    g.Components.Add(_currentRuler);
}
```

All this does is initialize each of your objects and add them to the game's component collection. You should include a call to this in your main `Game` object's `Initialize` method, so go back to your game1.cs file and add that now:

```
PerformanceHelper.InitializeWithGame(this);
```

Did you remember what else you should do here from earlier? If you want a good measure of your performance, you need to make sure you aren't artificially CPU or GPU bound, so add the following to your `Game` constructor:

```
graphics.SynchronizeWithVerticalRetrace = false;
this.IsFixedTimeStep = false;
```

Now your game is ready to run as fast it can! Go back to your `PerfHelper.cs` code file and add the following helper method:

```
public static void StartFrame()
{
    if (_firstFrame)
    {
        _firstFrame = false;
        _debugCommandUI.ExecuteCommand("tr on log:on");
        _debugCommandUI.ExecuteCommand("fps on");
    }
    _currentRuler.StartFrame();
}
```

On the first frame, execute a couple of commands: the first one to tell the system to turn on the time ruler with logging (tr stands for the time ruler) and the second one to tell the system to turn on the FPS counter. After that, you call the `StartFrame` method on your time ruler to signify the start of a new frame.

> **Note**
>
> You can actually expand the debug command UI with your own commands if you like, but that is beyond the scope of this chapter.

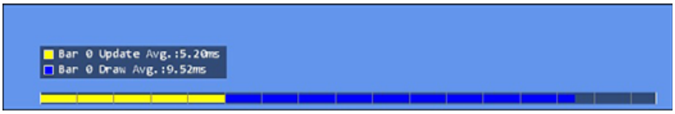
You also need access to the time ruler class externally, so add the following property accessor:

```
public static TimeRuler TimeRuler
{
    get
    {
        return _currentRuler;
    }
}
```

You need one more helper object to more easily use the time ruler, but before you add that to your code, let's take a quick moment to discuss what the time ruler actually does. At the most basic level, it draws colorful bars on screen that give you a rough idea of how long actions take. You can call the `BeginMark` method, passing in an index (a zero-based index to the colorful bar you want to draw on), a string that is any arbitrary name you want to use to describe the section of code you're about to run, and a color, which is the color this section is drawn in. You then have some of your own code execute, and you complete the cycle by calling `EndMark`, passing in the same index and name. Then, when the scene is finally rendered, you see colorful bars representing how long that code you wrote took to execute. For example, if you used code similar to this, you would see a single bar drawn at the bottom with the first portion colored yellow and the second portion colored blue:

```
PerformanceHelper.TimeRuler.BeginMark(0, "Update", Color.Yellow);
// Update Code
PerformanceHelper.TimeRuler.EndMark(0, "Update");
PerformanceHelper.TimeRuler.BeginMark(0, "Draw", Color.Blue);
// Draw Code
PerformanceHelper.TimeRuler.EndMark(0, "Draw");
```

The length of each bar represents how long the actions between the begin and end mark calls take along with the average time each action takes, much like you see in Figure 11.3.



🟨 Bar 0 Update Avg.:5.20ms
🟦 Bar 0 Draw Avg.:9.52ms

Figure 11.3 Rendering the time ruler with a single bar

However, if you use a different number for the index in the draw marker, say 1 instead of 0, you would see two different bars that show the same information, much like you see in Figure 11.4.

Notice how the second bar for the draw marker doesn't even start until the end of the first marker.

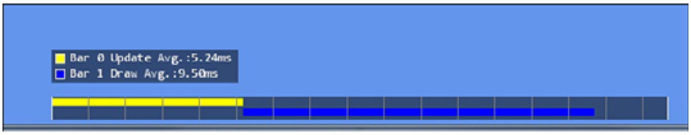


Figure 11.4 Rendering the time ruler with two bars

> **Note**
>
> Whatever index and name you pass in to `BeginMark` must be the same you pass in to `EndMark`.

Having to remember to use the `BeginMark` and `EndMark` calls everywhere, though, can be painful. What would be great is if you could create a simple helper object to encapsulate this code. If you remember the `using` statement from C#, that enables you to encapsulate code and it looks like a great opportunity to use here. All you need is an object that implements `IDisposable`, so add the following structure to your `perfhelper.cs` code file:

```
public struct TimeRulerHelper : IDisposable
{
    private int _index;
    private string _name;
    public TimeRulerHelper(int index, string name, Color c)
    {
        _index = index;
        _name = name;
        PerformanceHelper.TimeRuler.BeginMark(_index, _name, c);
    }
    public TimeRulerHelper(string name, Color c) : this(0, name, c)
    {
    }
    public void Dispose()
    {
        PerformanceHelper.TimeRuler.EndMark(_index, _name);
    }
}
```

Notice how you have this structure implemen `IDisposable`? During the construction of the object, `BeginMark` is called and during the Dispose method (for example, the end of the `using` block), the `EndMark` is called. This enables you to easily encapsulate the code you want to measure. You don't actually need both constructors, but the extra one is there in case you use only a single bar.

> **Note**
>
> Notice that this is a struct and not a class? You want this object to be a value type, so it is created on the stack and does not generate garbage, which ends up hurting the performance you're trying to measure!

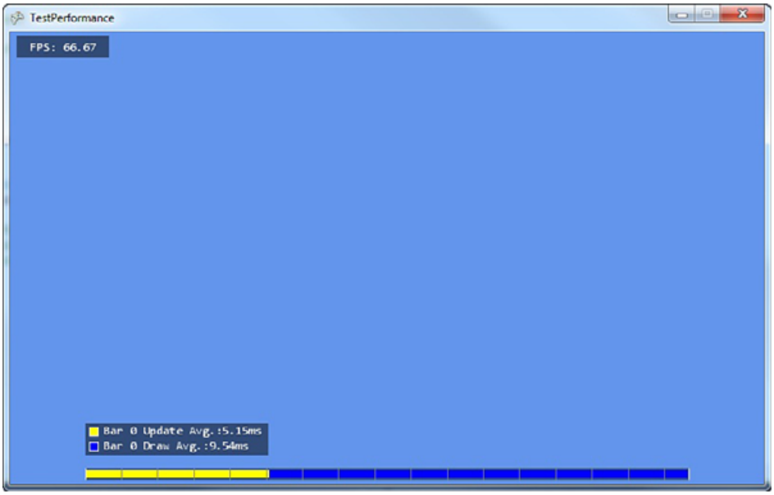
With that, you're now completely set up to start measuring performance in your game. To test out the time ruler code, let's go back to the `Game` and add the following to your `Update` method:

```
PerformanceHelper.StartFrame();
using (new TimeRulerHelper("Update", Color.Yellow))
{
    System.Threading.Thread.Sleep(5);
}
```

Of course, you aren't doing anything yet, but that `Sleep` call gives the illusion that you are. Next, add one for your `Draw` method:

```
using (new TimeRulerHelper("Draw", Color.Blue))
{
    System.Threading.Thread.Sleep(10);
}
```

Running the application now shows you the time ruler bar and the frame counter at the top, much like you see in Figure 11.5.



```
FPS: 66.67
```

```
Bar 0 Update Avg.:5.15ms
Bar 0 Draw Avg.:9.54ms
```

Figure 11.5 Your first experience with the time ruler

Let's take a few minutes to expand on this example to make it do something that is actually measurable. First, you need to add two models to your content project from the downloadable examples: `depthmodel.fbx` and `environmentmapmodel.fbx`. Now include this new class file in your project:

```
class ModelInformation
{
    public Model Model;
    public Vector3 Position;
    public Vector3 YawPitchRoll;
    public Vector3 YawPitchRollChange;
    public Vector3 Velocity;
}
```

This is basic information you want to store about your models, because you draw a lot of them, and you might as well have them move around so you have something interesting to look at. You also want to add a few local variables to your `Game` class:

```
const float BoxRange = 200.0f;
private List<ModelInformation> _modelList;
Matrix view;
Matrix proj;
Random r = new Random();
int bounceCheck = 0;
```

There is nothing you shouldn't recognize here. You store your models in the list, and you have your camera matrices, a random number generator, and a counter you use later. To generate a unique scene, add the following helper method to generate random vectors:

```
Vector3 GetRandomV3(float max)
{
    Vector3 v = new Vector3();
    v.X = (float)(r.NextDouble() * max);
    v.Y = (float)(r.NextDouble() * max);
    v.Z = (float)(r.NextDouble() * max);
    if (r.NextDouble() > 0.5)
        v.X *= -1;
    if (r.NextDouble() > 0.5)
        v.Y *= -1;
    if (r.NextDouble() > 0.5)
        v.Z *= -1;
    return v;
}
```

This generates a random vector within a certain maximum, and each component can be positive or negative. With that helper out of the way, you can now load your data by adding the following to the `LoadContent` method:

```
const int total = 100;
_modelList = new List<ModelInformation>();
for (int i = 0; i < total; i++)
{
    ModelInformation modelInfo = new ModelInformation();
    modelInfo.YawPitchRoll = GetRandomV3(1.0f);
    modelInfo.Position = GetRandomV3(BoxRange);
    modelInfo.Velocity = GetRandomV3(27.5f);
    modelInfo.YawPitchRollChange = GetRandomV3(5.1f);
    modelInfo.Model = (r.NextDouble() > 0.5) ?
        Content.Load<Model>("depthmodel") :
        Content.Load<Model>("environmentmapmodel");
    _modelList.Add( modelInfo );
}
view = Matrix.CreateLookAt(new Vector3(0, 3.0f, -250.0f), Vector3.Zero,
➥Vector3.Up);
proj = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
    GraphicsDevice.Viewport.AspectRatio, 1.0f, 1000.0f);
// turn on some lights
foreach (ModelMesh mm in Content.Load<Model>("depthmodel").Meshes)
{
    foreach (BasicEffect be in mm.Effects)
    {
        be.EnableDefaultLighting();
    }
}
foreach (ModelMesh mm in Content.Load<Model>("environmentmapmodel").Meshes)
{
    foreach (BasicEffect be in mm.Effects)
    {
        be.EnableDefaultLighting();
    }
}
```

Here you have a constant declared for the total number of models you want to render, so you can easily add more later to see how performance is affected. You randomly assign a position, velocity, and rotation to your model, and then pick randomly between the two models in the project. Finally, you set up your camera and turn on the default lighting on your two models. Now you want to see your models, so replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    using (new TimeRulerHelper("Draw", Color.Blue))
    {
        GraphicsDevice.DepthStencilState = DepthStencilState.Default;
        foreach (ModelInformation m in _modelList)
        {
            m.Model.Draw(Matrix.CreateFromYawPitchRoll(m.YawPitchRoll.X,
                m.YawPitchRoll.Y, m.YawPitchRoll.Z) *
                Matrix.CreateTranslation(m.Position), view, proj);
        }
    }
    base.Draw(gameTime);
}
```

Simply loop through each model and draw it. The one important thing to note here is setting the `DepthStencilState` before the drawing. The time ruler library turns off depth when it renders its text. It's generally a good idea to set any state you know you'll need before drawing your objects, and in this case, you need a depth buffer.

> **Note**
>
> You might also notice that the `base.Draw` method is still here now, and in some earlier examples, it was removed. If your game has components, you need to include the call to `base.Draw`. In earlier examples, when it was removed, no components were involved.

This isn't an exciting example so far! There are a lot of models, but they aren't doing much. You want them to occasionally bounce off each other, so you need a way to do that. Add the following helper method:

```
private bool IsColliding(ModelInformation m)
{
    foreach (ModelInformation info in _modelList)
    {
        if (m == info)
            continue;
        // Calculate difference
        Vector3 difference = info.Position - m.Position;
        float combinedRadius = m.Model.Meshes[0].BoundingSphere.Radius *
            m.Model.Meshes[0].BoundingSphere.Radius *
            info.Model.Meshes[0].BoundingSphere.Radius *
            info.Model.Meshes[0].BoundingSphere.Radius;
        // If they're colliding
        if (difference.LengthSquared() > combinedRadius  )
        {
            return true;
        }
    }
    return false;
}
```

Don't look at this method as a way to do collision detection. All it does is calculate the distance between the current model and all other models, and if any of the models are within the combined radius of itself and the model it is currently testing, it returns `true`, signifying that the model is colliding with something. Otherwise, it returns `false`.

With that, you can implement your `Update` method; replace your current one with the following:

```
protected override void Update(GameTime gameTime)
{
    PerformanceHelper.StartFrame();
    using (new TimeRulerHelper("Update", Color.Yellow))
    {
        System.Threading.Thread.Sleep(5);
        // Allows the game to exit
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();
        // Update model information
        foreach (ModelInformation m in _modelList)
        {
            m.Position += m.Velocity *
➥(float)gameTime.ElapsedGameTime.TotalSeconds;
            m.YawPitchRoll += m.YawPitchRollChange *
                (float)gameTime.ElapsedGameTime.TotalSeconds;
            // Check position
            if ((m.Position.X <= -BoxRange) || (m.Position.X >= BoxRange) ||
                (m.Position.Y <= -BoxRange) || (m.Position.Y >= BoxRange) ||
                (m.Position.Z <= -BoxRange) || (m.Position.Z >= BoxRange)  )
            {
                // Move back
                m.Position -= m.Velocity *
                    (float)gameTime.ElapsedGameTime.TotalSeconds;
                m.Velocity *= -1;
            }
            // If they're colliding, redirect in some random direction
            if ((bounceCheck % 10) == 0)
            {
                if (IsColliding(m))
                {
                    m.Velocity *= -GetRandomV3(1.0f);
                    // Start moving that way
                    m.Position += m.Velocity *
                        (float)gameTime.ElapsedGameTime.TotalSeconds;
                }
            }
            bounceCheck++;
        }
    }
    base.Update(gameTime);
}
```

Here you move the models based on their current velocity, and then check to see whether they've gone beyond a certain point, and if so, reverse the velocity (but first move it back to its original position). Then, every ten frames, you check the collisions with the helper methods you wrote a moment ago, and if it collides with something, you can randomly change the velocity. Now when you run the game, you see the objects flying around much like you see in Figure 11.6.

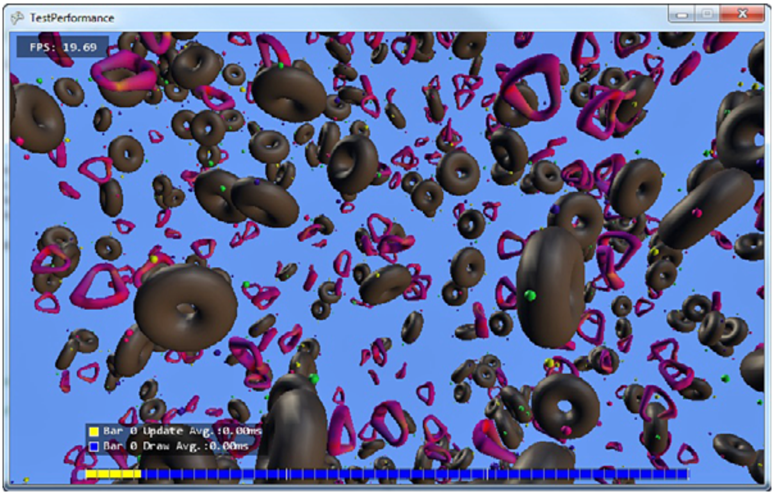


Figure 11.6 A simple example of measuring perf

With that out of the way, now we can spend a little time talking about various ways to detect if you're GPU bound, CPU bound, or if you have achieved a perfect balance (a great goal to strive for). There isn't a magic button you can press to determine this; you normally have to figure it out via trial and error.

With the time ruler, you can get a good idea of how long things take. If you notice that your `Update` method takes the majority of your time, you're probably CPU bound. You can add extra bars to help narrow down which portions of your methods taje the most amount of time. The time ruler can be used as an extremely flexible microbenchmark.

If you have a CPU profiler (which is discussed in slightly more detail in a few moments in this chapter), run that and check the amount of time spent in your `Update`, `Draw`, and `Present` methods. If the vast majority of your time is spent in `Update`, you are CPU bound! Otherwise, you need to check various things in your game.

If looking at your time ruler bars doesn't help you gauge where your performance problems lie, you can begin to narrow down where they are by changing your `Update` method. Add a small `Sleep(1)` call to the beginning of the Update method. If this doesn't affect your game's frame rate, then your game is most likely GPU bound (although you could be perfectly in sync).

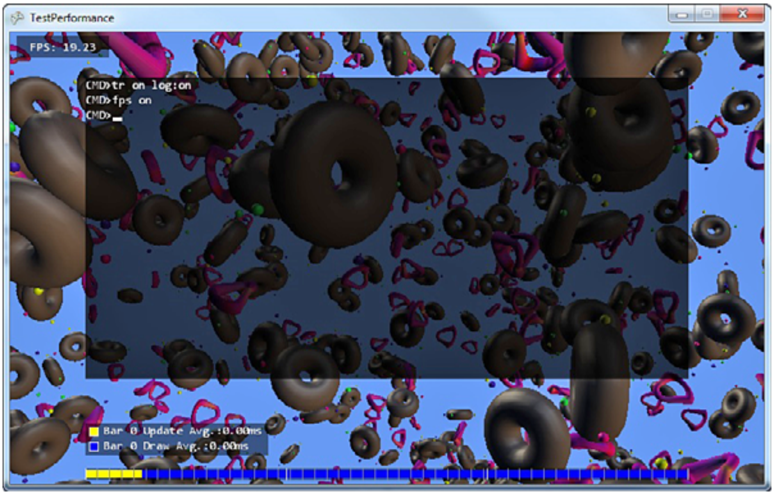
The Time Ruler Library

This chapter would be way too long if the entirety of the helper library was explained in detail. However, the entirety of the code for the library is included with the downloadable examples, and that code is well commented.

The `DebugManager` class is simply to hold the few graphics resources needed to render the text and the bars you see from the other components. The `FPSCounter` component is an easy-to-use and drop-in frame counter.

The `DebugCommandUI` is much more customizable than the others. Press the `Tab` key on your keyboard and you can see that it brings up a console screen that you can type in as seen in Figure 11.7. On Windows Phone, tap the upper left corner of the screen to bring up the console window and the upper right corner to bring up the on screen keyboard to type commands. This works on the Xbox, too, if you attach a keyboard.

At the time of this writing, you can read more from the author of this helper library at http://blogs.msdn.com/ito although be forewarned that the site is in Japanese.



```
FPS: 19.23
CMD>tr on log:on
CMD>fps on
CMD>_
Bar 0 Update Avg.:0.00ms
Bar 0 Draw Avg.:0.00ms
```

Figure 11.7 Seeing the command window console in the helper library

Performance Measurement Tools

A wide variety of tools are available for profiling. Depending on which flavor of Visual Studio you have, you might have one built in already (it doesn't exist in the free versions of Visual Studio). You can also use the free CLR profiler you can download from Microsoft's website.

Note

To use the CLR profiler with VS2010, you need to set the environment variable: set `COMPLUS_ProfAPI_ProfilerCompatibilitySetting=EnableV2Profiler`.

XNA Game Studio also comes with the Remote Performance Monitor tool that you can use to help profile your game on Xbox.

Profilers can be an enormous help in discovering what is going on in your game and you should certainly spend some time investigating them.

Cost of Built-In Shaders

A point that has been made several times in this chapter is that you should know what your code costs. In this section, we discuss the basic costs of the five built-in effects that are included with the runtime. Because it is hard to say unequivocally what the cost of these will be at runtime, instead you see the number of instructions for each permutation of the shaders with the basic high-level concept that something that is more simple (fewer instructions) is going to be faster. Of course, spending all of your time choosing the fewest instructions is for naught if you aren't GPU bound!

`BasicEffect` has a large number of different permutations that it can use depending on which properties you've chosen. However, it can be broken down into four main versions, as listed in Table 11.1.

Table 11.1 **Cost of BasicEffect**

	Vertex Shader	Pixel Shader
LightingEnabled =false	5	1
One Vertex Light	40	1
Three Vertex Lights	60	1
PreferPerPixelLighting=true	18	50
TextureEnabled=true	+1	+2
FogEnabled=true	+4	+2

The items that have a cost are represented as plus; some other number means that those features simply cost many more instructions on top of whatever other options you've chosen. Notice that aside from when you are using per pixel lighting, all of the pixel shader instruction counts are very low, but when you turn on per pixel lighting, it uses quite a few instructions. If you are GPU bound and are bound by the pixel pipeline, then you certainly wouldn't want to use this option!

The costs for the `SkinnedEffect` object can be found in Table 11.2.

Table 11.2 **Cost of SkinnedEffect**

	Vertex Shader	Pixel Shader
One Vertex Light	55	4
Three Vertex Lights	75	4
PreferPerPixelLighting=true	33	51
WeightsPerVertex=2	+7	+0
WeightsPerVertex=4	+13	+0
FogEnabled=true	+0	+2

The first three rows describing the cost of the lighting are what the costs are when the `WeightsPerVertex` is set to 1; however, the *default* value is 4, so you need to take this into account when looking at these options. You might notice that unlike `BasicEffect`, you cannot turn off lighting completely or disable texturing in `SkinnedEffect`.

The costs for the `EnvironmentMapEffect` object can be found in Table 11.3.

Table 11.3 **Cost of EnvironmentMapEffect**

	Vertex Shader	Pixel Shader
One Vertex Light	32	6
Three Vertex Lights	36	6
FresnelFactor != 0	+7	+0
EnvironmentMapSpecular != 0	+0	+2
FogEnabled=true	+0	+2

Notice that when fewer features are supported by an effect, the cost decreases as you would expect. `DualTextureEffect` has the simplest effect (and potentially lowest cost), as you can see in Table 11.4.

Table 11.4 **Cost of DualTextureEffect**

	Vertex Shader	Pixel Shader
Default	7	6
FogEnabled=true	+4	+2

Note that if you are pixel pipeline bound, `DualTextureEffect` does have the potential for a larger amount of pixel instructions than most permutations of `BasicEffect`, although `DualTextureEffect` is probably worth the cost to create some compelling visuals on the phone. The last effect, `AlphaTestEffect,` has a couple or permutations depending on which `AlphaFunction` it is used with, as seen in Table 11.5.

Table 11.5 **Cost of AlphaTestEffect**

	Vertex Shader	Pixel Shader
<, <-, >=, >	6	6
==, !=	6	10
FogEnabled=true	+4	+2

We hope this gives you the extra information you need to get a better grasp on how much the effects that ship with the runtime cost for your game.

Summary

Performance is a tricky subject. If your game runs well enough, maybe you don't need to worry about it at all! Hopefully, throughout this chapter, you learned good practices for dealing with performance and now know ways to find potential pitfalls, measure your game's performance, and fix issues you run into!

The next chapter delves into adding interactivity to your games by allowing user input!

This page intentionally left blank

12

Adding Interactivity with User Input

Interactivity is the distinguishing factor between games and other forms of entertainment. User involvement and interaction is what makes a game different. Games allow for interactivity by using some type of user input, such as a keyboard or a gamepad, and then changing what is displayed on the screen based on that input. This chapter focuses on how to access the user input from a number of devices.

In this chapter, you learn:

- Access input data from the keyboard, mouse, and gamepad
- Handling multitouch screen input and gestures
- Reading accelerometer and location service data

Using Input in XNA Game Studio

Up to this point, we have demonstrated how to display fancy two-dimensional (2D) and three-dimensional (3D) graphics for your games. It is time to make your games truly interactive. Without user input, your game is more of a screen saver or video than it is a game. Designing and implementing good user input enables the game player to feel immersed in the virtual world you create. Throwing together poorly designed and implemented user input leaves players frustrated and causes controllers to be thrown across the room.

Remember that XNA Game Studio 4.0 supports Windows, Xbox 360, and Windows Phone. XNA Game Studio provides the capability to interact with a large number of different input devices across all of these platforms. We cover how to poll and understand the state of all devices including the keyboard, mouse, Xbox 360 gamepad and controllers, and multitouch screens. Although not directly part of XNA Game Studio, we also cover the use of Windows Phone-specific features including accelerometer, location service, and phone vibration.

> **Digital and Analog Input**
>
> There are two types of values when dealing with user input analog and digital. Digital input allows for two specific states. Examples of digital input are the keys on a keyboard or the buttons on the Xbox 360 gamepad. The keys or the buttons are either pressed or they are released. Analog input allows for a much wider range of values often expressed in float values from –1 to 1 or 0 to 1. Examples of analog input are the mouse location or the thumb stick location on the Xbox 360 gamepad.

Polling versus Event-Based Input

There are generally two mechanisms to handle user input polling and input events. With polling, input devices are queried or polled for their current state. The state is then preserved in some type of object such as `KeyboardState` or `MouseState`. These states do not change as the input device itself changes. To get the latest state, the input device needs to be polled again. Often the states of the current and previous frames are required when handling user input utilizing polling.

Because of the game loop nature of update and draw that games utilize, XNA Game Studio APIs utilize polling methods to query the current state of each of the input devices. The input devices are generally polled at the start of the update phase of the game loop. The user input along with the elapsed time that has occurred since the last update call is then used in game play logic such as moving the players' characters.

Event-based input systems are utilized more in application development. For example, within a Silverlight application, you can create an event that fires when the mouse is clicked. The event code fires only when that condition occurs and your code does not need to poll the mouse. This works well in applications that utilize an event-based application model and generally don't have continuous loops running as in games.

The Many Keys Of A Keyboard

The keyboard is one of the most widely used input devices for games. Because almost every Windows PC has a keyboard along with the mouse, they are the standard input devices for Windows PC gaming. With well over 100 keys, the keyboard has the most input buttons of any input device supported by XNA Game Studio. You use keyboards in games for controlling movement and for game scenarios with a large number of actions. For example, in a real-time strategy game, a player can use the keyboard to issue commands to units in the game much quicker than if the player had to click an onscreen icon.

Keyboard input is not limited to just the Windows platform. XNA Game Studio also supports reading keyboard input on Xbox 360 and Windows Phone. The Xbox 360 supports two types of keyboard devices. Users can plug USB keyboards directly into Xbox 360 as they would an Xbox 360 gamepad. The USB keyboard then works normally as if it were attached to a Windows PC. The other keyboard available on the Xbox 360 is the chatpad. The chatpad is a mini keyboard that attaches directly to the bottom of an Xbox 360 gamepad.

> **Note**
>
> Chatpad input is available only on Xbox 360. Although Xbox 360 controllers work on Windows, the chatpad keyboard input does not.

Most players do not have a chatpad or keyboard attached to their Xbox 360, so when developing your game for Xbox 360, you should not require these input devices.

XNA Game Studio also supports keyboard input on some Windows Phone devices. Some Windows Phone devices come with a hardware keyboard. Not all devices have a hardware keyboard, so your game should not require keyboard input if targeting the Windows Phone platform.

Reading Keyboard State

As we have discussed, reading input devices in XNA Game Studio is a matter of requesting the current state of the device. That state can then be used for updating actions in your game. In XNA Game Studio, the keyboard is represented by the `Keyboard` class. The main method the `Keyboard` class contains that we use is `Keyboard.GetState()`. This method returns the current state of the keyboard. The keyboard state is stored in the `KeyboardState` structure. Table 12.1 contains the methods exposed by the `Keyboard` type.

Table 12.1 **Methods of KeyboardState**

Method	Description
`GetPressedKeys`	Returns array of keys that are pressed
`IsKeyDown`	Returns true if the provided key is currently pressed
`IsKeyUp`	Returns true if the provided key is not currently pressed

To read the current state of the keyboard, use the following line of code:

```
KeyboardState currentKeyboardState = Keyboard.GetState();
```

> **Reading Xbox 360 Chatpad Input**
>
> Because each Xbox 360 gamepad might have a chatpad attached, it is possible that more than one player can enter keyboard input. To read the keyboard input for a specific player, use the `Keyboard.GetState(PlayerIndex playerIndex)` overload and specify the player index.

Now that the current state of the keyboard is saved into the `KeyboardState` structure, you can read what keys were pressed at the time that `Keyboard.GetState()` was called. `KeyboardState` contains a number of methods to determine the current state of a specific key. To check whether a specific key is currently pressed, use the `KeyboardState.IsKeyDown(Keys key)` method. This method takes an enum value of the type `Keys`. The `Keys` enum contains all of the readable keyboard keys using XNA Game Studio. `Keys` contains values for normal alphanumeric keys you expect such as the letter A or the number 1. The `Keys` enum also contains values for more rarely used keys that exist only on some keyboards such as the application launch and media buttons. To determine whether a specific key is currently pressed, use the following lines of code:

```
if (currentKeyboardState.IsKeyDown(Keys.A))
{
    // The A button on the keyboard is pressed
}
```

> **Note**
>
> User input and handling often, but not always, occurs in the game's `Update` method. Reading the current state of an input device can happen anywhere in your game code.

If the player presses the `A` key over multiple frames, the `IsKeyDown` method will continue to return true. This means that the previous bit of code continues to be true and does not require that the player press the key each frame. `IsKeyDown` returns true if the key was pressed when the state was polled using `Keyboard.GetState`. It does not express if the key was pressed in a previous frame. This can become a problem because often in games, it is desirable for an action to happen only once per key press. If you want an action to happen only once per key press, you need to store the `KeyboardState` in a previous frame.

In your game class, define the following member variable:

```
KeyboardState lastKayboardState;
```

At the end of the `Update` method where you have read the current state of the keyboard, set the last frame state to the current state:

```
lastKeyboardState = currentKeyboardState;
```

This preserves the previous frame's keyboard state so you can use it in the following frame. To determine whether a key is pressed in the frame, use the `IsKeyDown` method along with the `IsKeyUp` method exposed by `KeyboardState`:

```
if (currentKeyboardState.IsKeyDown(Keys.A) && lastKeyboardState.IsKeyUp(Keys.A))
{
    // The A button was pressed this frame
}
```

Because you know the key is pressed in this frame and is not pressed in the previous frame, you know it is pressed in this frame for the first time and you can execute the appropriate action.

`KeyboardState` provides an indexer overload that enables you to determine the `KeyState` for a particular key. The following lines of code check the state of the spacebar key:

```
KeyState spacebarState = currentKeyboardState[Keys.Space];
if (spacebarState == KeyState.Down)
{
    // Spacebar is pressed down this frame
}
```

```
else if (spacebarState == KeyState.Up)
{
    // Spacebar is up this frame
}
```

`KeyboardState` provides the capability to return an array of the `Keys` pressed. The following lines of code get an array of `Keys` and loops over them:

```
Keys[] pressedKeys = currentKeyboardState.GetPressedKeys();
foreach (Keys key in pressedKeys)
{
    // Game logic to handle each key
}
```

Moving Sprite Based on Keyboard Input

Now that we covered how to read the current state of the keyboard and can determine whether a key is pressed or released, let's move a sprite around the screen using the keyboard. First, you need to declare some member variables to store your sprite texture, the sprites position, and the last keyboard state:

```
Texture2D spriteTexture;
Vector2 spritePosition;
float spriteSpeed = 100f;
KeyboardState lastKeyboardState;
```

Next, load a texture to use as your sprite. In the game's `LoadContent` method, add the following line of code:

```
spriteTexture = Content.Load<Texture2D>("XnaLogo");
```

Now, you are ready for some keyboard input. In the game's `Update` method, you need to store the current state and move the sprite's position based on the arrow keys, as follows:

```
// Store the current state of the keyboard
KeyboardState currentKeyboardState = Keyboard.GetState();

// Move the sprite position based on keyboard input
if (currentKeyboardState.IsKeyDown(Keys.Left))
    spritePosition.X -= spriteSpeed *
➥(float)gameTime.ElapsedGameTime.TotalSeconds;
if (currentKeyboardState.IsKeyDown(Keys.Right))
    spritePosition.X += spriteSpeed *
➥(float)gameTime.ElapsedGameTime.TotalSeconds;
if (currentKeyboardState.IsKeyDown(Keys.Up))
    spritePosition.Y -= spriteSpeed *
➥(float)gameTime.ElapsedGameTime.TotalSeconds;
```

```
if (currentKeyboardState.IsKeyDown(Keys.Down))
    spritePosition.Y += spriteSpeed *
➥(float)gameTime.ElapsedGameTime.TotalSeconds;

// Store the keyboard state for next frame
lastKeyboardState = currentKeyboardState;
```

Whenever the arrow keys are pressed, you move the sprite in the appropriate direction. Notice that "up" corresponds to moving the sprite in the negative Y direction because values for Y increase when moving down the screen.

When you move the sprite, you don't move by a constant value. Take the elapsed game time into account. Each frame can take a different amount of time to complete. If a constant movement amount is used to update the position of the sprite, it might appear to shudder as it moves across the screen. Correct this using the `GameTime.ElapsedGameTime` property. You previously used the elapsed time in total seconds. This means pressing the right arrow for one second moves the sprite 100 pixels to the right no matter how many frames happen over that second.

> **Note**
>
> Older PC video games didn't always take into account the elapsed time when animating and moving sprites. As computing power of PCs increased, these games were able to run faster and so did the animations and speed of the games, making some of the game unplayable.

Finally, you need to display the sprite on the screen using the position you have been updating. To display the sprite, add the following lines of code to your game's `Draw` method:

```
spriteBatch.Begin();
spriteBatch.Draw(spriteTexture, spritePosition, Color.White);
spriteBatch.End();
```

Now if you run your game, you can control the sprite on the screen using the keyboard arrow keys. Congratulations—you have created your first XNA Game Studio interactive game. This simple game should look similar Figure 12.1.

Onscreen Keyboard

If your goal is to gather user text input, the `Keyboad` class is not the best solution. If you try to implement text entry using the `Keybord` state APIs, you run into a number of issues such has handling the current shift state of the keyboard and making sure to call `GetState` quick enough that you don't miss a single key press. The `Keyboad` API is not designed for text entry (see Figure 12.2).

For text entry, use the `Guide.BeginShowKeyboardInput` method. If you are familiar with C#, notice that `BeginShowKeyboardInput` is an asynchronous method. This means that when you ask to show the keyboard, this method quickly returns but that does not mean the keyboard has been shown or that the user has entered text into the keyboard. For your game to continue running, it needs to handle a method that takes a number of frames to return. `BeginShowKeyboardInput` returns an `IAsyncResult` object. This object

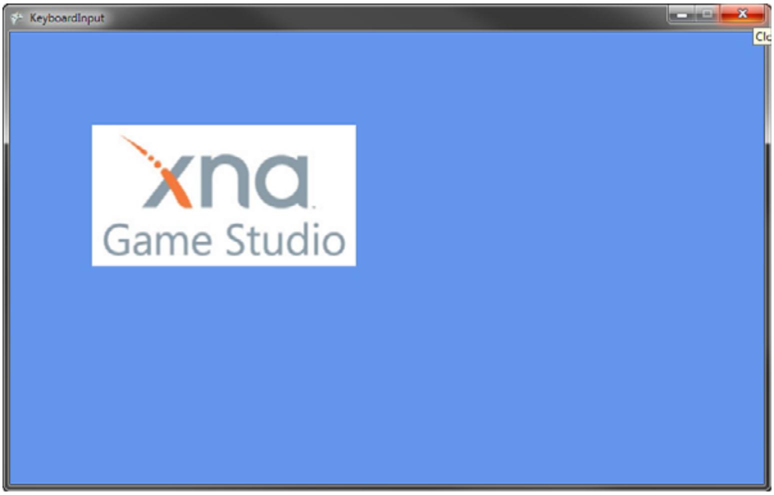

KeyboardInput
Game Studio

Figure 12.1 Sprite controlled by user keyboard input

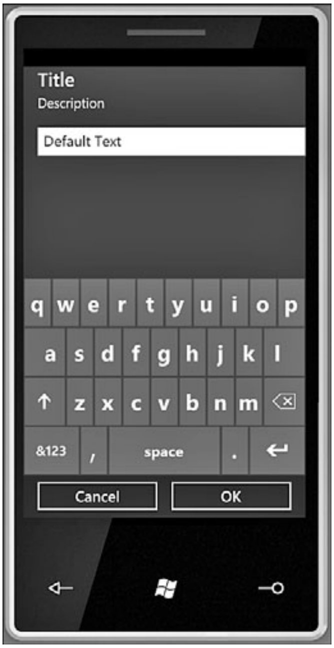

Title
Description
Default Text
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
&123
,
space
.
Cancel
OK

Figure 12.2 Onscreen keyboard in the Windows Phone emulator

is used each frame to check on the state of the call to `BeginShowKeyboardInput`. If the `IsCompleted` property of the result is true, that means the call has completed and that you can request the string from the keyboard input. This is done calling `Guide.EndShowKeyboardInput` passing in the result. The method returns the string the user entered. The string might be null because the user can cancel the onscreen keyboard.

To add the onscreen keyboard to your game, store the async result over multiple frames. You also need to store the string returned. Add the following members to your game class:

```
IAsyncResult keyboardAsyncResult;
string message = "";
```

We cover the `Guide.BeginShowKeyboardInput` method, sometimes called gamer services, later in the chapter. The important point to know now is that your game constructor needs to initialize gamer services on Windows and Xbox 360 to use the onscreen keyboard. This is done by adding the `GamerServicesComponent` to the `Components` list of your game. On Windows and Xbox, add the following line of code at the end of your game's constructor:

```
Components.Add(new GamerServicesComponent(this));
```

Next, check for a spacebar press. If the user presses the spacebar and the onscreen keyboard is not already visible, you start to display the keyboard. Add the following lines of code to your game's `Update` method:

```
if (keyboardAsyncResult == null &&
    currentKeyboardState.IsKeyDown(Keys.Space) &&
➥lastKeyboardState.IsKeyUp(Keys.Space))
{
    keyboardAsyncResult = Guide.BeginShowKeyboardInput(PlayerIndex.One,
                                                        "Title",
                                                        "Description",
                                                        "Default Text",
                                                        null, null);
}
```

In the previous code, check the keyboard async result to make sure you are not already waiting for a previous call. Then, if the spacebar is pressed, you make a call to `BeginShowKeyboardInput`. The first parameter is the `PlayerIndex` you want to have control of the onscreen keyboard. This can be only `PlayerIndex.One` for the Windows and Windows Phone platforms, but on Xbox 360, up to four players can play a game and you might want player three to enter his or her character's name. In this case, player three should be the one controlling the onscreen keyboard. The next three parameters are `string` values representing the title, description, and default text to display. We cover the last two values a little later.

Now that you asked for the keyboard to display, it should display when the user presses the spacebar. But when a user enters text or cancels the keyboard, the return value is thrown away. To get the result of what the user enters into the keyboard, you need to call `Guide.EndShowKeyboardInput`. Calling this method blocks until the previous call to `BeginShowKeyboardInput` completes, so first check whether the call is complete. After the code you just added, place the following lines of code:

```
// Check to see if the result has completed
if (keyboardAsyncResult != null && keyboardAsyncResult.IsCompleted)
{
    message = Guide.EndShowKeyboardInput(keyboardAsyncResult);
    if (message == null)
    {
        // String is null since the user canceled the input
    }
    keyboardAsyncResult = null;
}
```

Notice that the `EndShowKeyboardInput` call takes the `IAsyncResult` from the `BeginShowKeyboardInput` call and returns the string entered by the user. This string can be null if the user canceled the onscreen keyboard.

In the previous code, you use the `IAsyncResult` from the `BeginShowKeyboardInput` call to check whether the call completed each frame. Instead of checking the status of each frame, you can use an `AsyncCallback` method that is called after the method completes. To use the callback method, add the following method to your game:

```
void StoreKeyboardResult(IAsyncResult result)
{
    message = Guide.EndShowKeyboardInput(result);

    if (message == null)
    {
        // String is null since the user canceled the input
    }
}
```

After the `BeginShowKeyboardInput` is complete and the user has entered text, this method is called. Similar to the previous example, you then call `EndShowKeyboardInput` and check the resulting string. To have `BeginShowKeyboardInput` use the callback, change the call to look like the following:

```
if (currentKeyboardState.IsKeyDown(Keys.Space) &&
➥lastKeyboardState.IsKeyUp(Keys.Space))
{
    keyboardAsyncResult = Guide.BeginShowKeyboardInput(PlayerIndex.One,
                                                        "Title",
                                                        "Description",
                                                        "Default Text",
                                                       StoreKeyboardResult, null);
}
```

You no longer need to store the `IAsyncResult` because the `StoreKeyboardResult` method is called after the user has completed entering text.

The last parameter of `BeginShowKeyboardInput` is used to pass in an object to be stored in the `IAsyncResult.AsyncState` property. This can be useful when you need a specific object after the operation is completed. For example, if you are asking a user to enter his or her character name and you want to store the result in your game's specific `Character` class, you can pass the user's instance of that `Character` class into `BeginShowKeyboardInput` and when the call completes, use the `IAsyncResult.AsyncState` property to access the instance of the `Character` class.

Precision Control of a Mouse

The mouse is one of the most accurate input devices supported by XNA Game Studio. It is used for a variety of tasks in games from selecting units and giving orders in a real-time strategy game to targeting and shooting weapons in first-person shooters and selecting moves in a chess game. Although the mouse APIs are available across all XNA Game Studio platforms, they don't behave the same across all platforms. On Windows, the mouse APIs perform as you expect giving you the current state of the physical mouse plugged into your PC. On Xbox 360, there is never mouse input because the Xbox 360 does not support mouse input. In this case, the mouse state always contains default values with a position at 0, 0 and no buttons pressed. On Windows Phone, there is also no physical mouse device, but because all Windows Phone devices must support multitouch, the mouse state is returned as the location and state of the first finger detected on the multitouch screen.

Reading Mouse State

Reading the current mouse state is similar to reading keyboard state. The `Mouse` class provides a static `GetState` method that returns the current mouse state stored in a `MouseState` structure. Table 12.2 contains the properties exposed by the `MouseState` structure.

Table 12.2 **Properties of MouseState**

Property	Type	Description
`X`	`int`	The horizontal position of the mouse
`Y`	`int`	The vertical position of the mouse
`ScrollWheelValue`	`int`	The scroll position of the mouse wheel
`LeftButton`	`ButtonState`	State of the left mouse button
`RightButton`	`ButtonState`	State of the right mouse button
`MiddleButton`	`ButtonState`	State of the middle mouse button
`XButton1`	`ButtonState`	State of extra mouse button 1
`XButton2`	`ButtonState`	State of extra mouse button 2

To read the current mouse state, add the following line of code to your game's `Update` method:

```
MouseState currentMouseState = Mouse.GetState();
```

> **Note**
>
> The `GetState` method supports reading only one mouse device, unlike the `Keyboard.GetState` method that we discussed previously.

Now that you have stored the current state of the mouse, you should also store the previous frame's mouse state in the same manner as you did for the keyboard. You can use this previous state to determine whether a button is clicked in a specific frame or held down over multiple frames. Declare the following member variable in your game:

```
MouseState lastMouseState;
```

At the end of your game's `Update` method, add the following line of code to store the previous frame's mouse state:

```
lastMouseState = currentMouseState;
```

The `MouseState` structure contains both analog data for the mouse and scroll wheel position and digital data for the mouse buttons. To retrieve the current position of the mouse, use the `MouseState.X` and `MouseState.Y` properties. These properties return the current position of the mouse on the screen. On Windows, if you are in windowed mode, the mouse can travel outside of the client bounds of the window. In this case, the mouse position is relative to the origin of the window's client area in the top left corner. If your mouse moves outside the client area, you receive negative values or values outside of the size of the client area depending on the direction you move outside of the client bounds. To store the current mouse position, use the following lines of code:

```
// Store the current mouse position
Vector2 mousePosition;
mousePosition.X = currentMouseState.X;
mousePosition.Y = currentMouseState.Y;
```

Along with the mouse position, you can also read the current mouse wheel using the `MouseState.ScrollWheelValue` property. This is an integer value of the current mouse wheel position. This position is accumulated while your game runs. If you want to determine whether the wheel is scrolled on this frame, you need to check whether the value in the current frame is greater or less than the previous frame. Use the following code to determine whether the mouse scroll wheel is scrolled forwards or backwards in the current frame:

```
// Determine if the mouse wheel was scrolled
if (currentMouseState.ScrollWheelValue > lastMouseState.ScrollWheelValue)
{
    // The mouse scroll wheel was scrolled forwards
}
```

```
else if (currentMouseState.ScrollWheelValue < lastMouseState.ScrollWheelValue)
{
    // The mouse scroll wheel was scrolled backwards
}
```

The `MouseState` structure contains five button properties to determine whether up to five different buttons on the mouse are pressed. Not all mice have five buttons. Similar to the keyboard's `KeyState`, the `ButtonState` returned by these properties has two values. In the case of the button, these values are `Pressed` and `Released`. To determine whether the left mouse button is pressed, use the following lines of code:

```
// Determine if the left mouse button is pressed
if (currentMouseState.LeftButton == ButtonState.Pressed)
{
    // Left mouse button is pressed
}
```

Similar to the keyboard, if you want to determine whether a button is clicked in the current frame and has not been held down, you need to also check the previous frame's mouse state and verify that the button is not pressed.

```
// Determine if the left mouse button was pressed this frame
if (currentMouseState.LeftButton == ButtonState.Pressed &&
    lastMouseState.LeftButton == ButtonState.Released)
{
    // Left mouse button was pressed for the first time this frame
}
```

> **Note**
>
> By default on Windows, the mouse is hidden when it is over the client area of the window. To view the mouse position while it is over the client area of the window, set the `IsMouseVisible` property of your game class to true.

Moving Sprite Based on Mouse Input

Now that you know how to retrieve the current state of the mouse device, let's use this state to move, rotate, and flip a sprite on the screen. First, you need to declare the following member variables to store the sprite texture, position, rotation, and if the sprite is flipped horizontally or vertically on the screen:

```
Texture2D spriteTexture;
Vector2 spritePosition;
Vector2 spriteCenter;
MouseState lastMouseState;
float rotation;
SpriteEffects flipOptions;
```

Next, load a texture to use as your sprite and set the sprite's center based on the texture width and height. In the game's `LoadContent` method, add the following lines of code:

```
spriteTexture = Content.Load<Texture2D>("XnaLogo");
spriteCenter.X = spriteTexture.Width / 2.0f;
spriteCenter.Y = spriteTexture.Height / 2.0f;
```

Now, update the current position, rotation, and flip values based on the current input of the mouse. In your game's `Update` method, add the following lines of code:

```
// Get the current mouse state
MouseState currentMouseState = Mouse.GetState();

// Store the current sprite position
spritePosition.X = currentMouseState.X;
spritePosition.Y = currentMouseState.Y;

// Flip the sprite horizontally if the left button is pressed
if (currentMouseState.LeftButton == ButtonState.Pressed &&
    lastMouseState.LeftButton == ButtonState.Released)
{
    flipOptions ^= SpriteEffects.FlipHorizontally;
}

// Flip the sprite vertically if the right button is pressed
if (currentMouseState.RightButton == ButtonState.Pressed &&
    lastMouseState.RightButton == ButtonState.Released)
{
    flipOptions ^= SpriteEffects.FlipVertically;
}

// Rotate the sprite if the mouse wheel has been scrolled
if (currentMouseState.ScrollWheelValue > lastMouseState.ScrollWheelValue)
{
    rotation += (float)gameTime.ElapsedGameTime.TotalSeconds;
}
else if (currentMouseState.ScrollWheelValue < lastMouseState.ScrollWheelValue)
{
    rotation -= (float)gameTime.ElapsedGameTime.TotalSeconds;
}
// Store the last frames state
lastMouseState = currentMouseState;
```

Now that you updated the sprite's position, rotation, and flip values, you can draw the sprite on the screen. In your game's `Draw` method, add the following lines of code:

```
// Draw the sprite
spriteBatch.Begin();
spriteBatch.Draw(spriteTexture, spritePosition, null, Color.White,
                 rotation, spriteCenter, 1.0f, flipOptions, 0);
spriteBatch.End();
```

The sprite now draws on the screen and updates its position, rotation, and flip, responding to mouse input (see Figure 12.3).

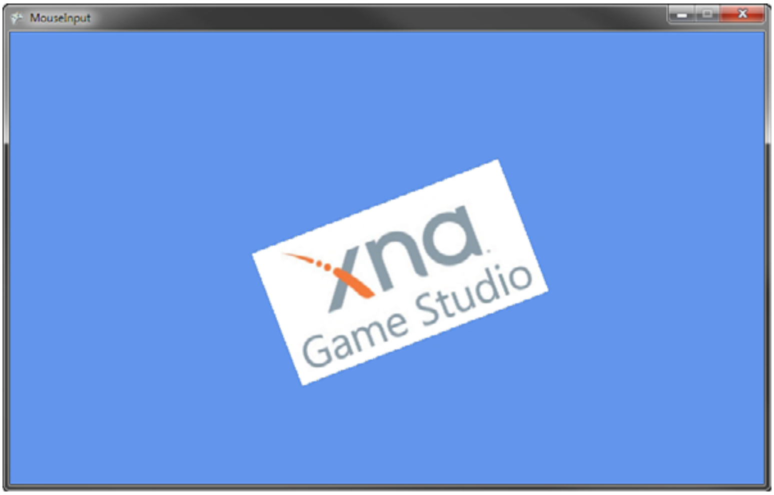


Figure 12.3 Using mouse input to move and rotate sprite

Setting the Mouse Position

Sometimes it is necessary to set the current position of the mouse on the screen. To set the position of the mouse, use the static `Mouse.SetPosition` method. This method takes the screen space integer X and Y values. To set the mouse to a location with the values of X: 400 Y: 240, use the following line of code:

```
Mouse.SetPosition(400, 240);
```

> **Setting the Mouse Window Handle**
>
> On Windows, the mouse state depends on the window it is attached to. By default, the mouse is set to be the game window that is created by your `Game` class. You might have to set the window manually if you are creating tools that use multiple windows. In this case, set the `Mouse.WindowHandle` property to the window you want the mouse attached to.

Xbox 360 Gamepad

The Xbox 360 gamepad is the controller that is included when you purchase an Xbox 360. It comes in both a wired and wireless model with several colors available to choose from (see Figure 12.4).

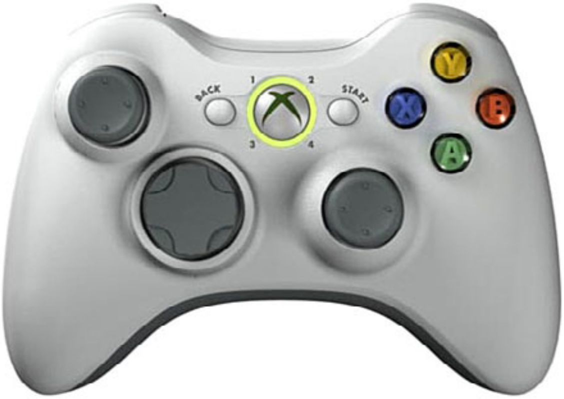


Figure 12.4 Xbox 360 wireless controller

The Xbox 360 gamepad features several analog and digital inputs, including face buttons, a direction pad, two thumb sticks, two triggers, and the circular Xbox 360 guide button in the center. The guide button is reserved and input from it is not available through the XNA Game Studio APIs. Along with input, the Xbox 360 gamepad also offers the capability to give force feedback to the user in the form of two vibration motors in the controller. The Xbox 360 gamepad can be used with a Windows PC. Wired controllers can plug directly into the PC using the USB connection. Wireless gamepads require a PC wireless receiver designed for the Xbox 360 wireless controller. Windows Phone does not support the Xbox 360 gamepad.

> **Note**
>
> When you create a new XNA Game Studio project, look at the `Update` method. It contains code that checks whether the `Back` button has been pressed and if it has to call exit on the game.
>
> Although Windows Phone does not support the Xbox 360 gamepad, there is one exception. The physical back button on Windows Phones is reported as the `Back` button on the controller for `PlayerIndex.One`.

Reading Gamepad State

Just as with keyboard and mouse input, the gamepad input is read by polling the current state of a gamepad. The `GamePad.GetState` method is used to return the `GamePadState` structure, which can then be used to determine what buttons are currently pressed and the location of the thumb sticks and triggers. Tables 12.3 and 12.4 contain the methods and properties exposed by the `GamePadState` structure.

Table 12.3 **Methods of GamePadState**

Method	Description
`IsButtonDown`	Returns true if the provided button is currently pressed
`IsButtonUp`	Returns true if the provided button is not currently pressed

Table 12.4 **Properties of GamePadState**

Property	Type	Description
`Buttons`	`GamePadButtons`	Structure that contains the state of all gamepad buttons
`DPad`	`GamePadDPad`	Structure that contains the state of all DPad buttons
`ThumbSticks`	`GamePadThumbSticks`	Structure that contains the position values for left and right thumb sticks
`Triggers`	`GamePadTriggers`	Structure that contains the position of the left and right triggers
`IsConnected`	`bool`	Returns true if the gamepad is connected
`PacketNumber`	`int`	Identifier

To read the current state of a gamepad, add the following line of code to your game's `Update` method:

```
GamePadState currentGamePadState = GamePad.GetState(PlayerIndex.One);
```

The `GetState` method requires a `PlayerIndex`. In this case, we request the first player's gamepad state. The Xbox 360 gamepad features four lights around the center guide button that displays the gamepads current `PlayerIndex`.

Reading Gamepad Buttons

Now that you have the current state of the gamepad, you can check whether any of the buttons are pressed by using the `GamePadState.Buttons` property. Buttons returns a `GamePadButtons` structure, which contains properties for all of the buttons supported by gamepads. These properties all are of the `ButtonState` type, which is the same used for the mouse buttons. Table 12.5 contains the list of properties exposed by the `GamePadButtons` structure.

Table 12.5 **Properties of GamePadButtons** **Continued**

Property	Type	Description
`A`	`ButtonState`	Pressed state of the A button
`B`	`ButtonState`	Pressed state of the B button
`X`	`ButtonState`	Pressed state of the X button

Table 12.5 **Properties of GamePadButtons Continued**

Property	Type	Description
`Y`	`ButtonState`	Pressed state of the Y button
`Back`	`ButtonState`	Pressed state of the Back button
`Start`	`ButtonState`	Pressed state of the Start button
`LeftShoulder`	`ButtonState`	Pressed state of the Left Shoulder button
`RightShoulder`	`ButtonState`	Pressed state of the Right Shoulder button
`LeftStick`	`ButtonState`	Pressed state of the Left Stick button
`RightStick`	`ButtonState`	Pressed state of the Right Stick button
`BigButton`	`ButtonState`	Pressed state of the Big Button

> ### Gamepad State Packet Numbers
>
> If you call `GamePad.GetState` in a tight loop, you might receive the same state more than once. If you are concerned that you might call `GetState` too quickly, check the `PacketNumber` property on each `GamePadState` to verify that the numbers are different. States with the same `PacketNumber` represent that the device state has not updated.

To determine whether the `GamePad` buttons are pressed, add the following lines of code to your game:

```
// Determine if the A button has been pressed
if (currentGamePadState.Buttons.A == ButtonState.Pressed)
{
    // A button is pressed
}
if (currentGamePadState.Buttons.Start == ButtonState.Pressed)
{
    // Start button is pressed
}
if (currentGamePadState.Buttons.RightShoulder == ButtonState.Pressed)
{
    // Right shoulder button is pressed
}
if (currentGamePadState.Buttons.LeftStick == ButtonState.Pressed)
{
    // Left stick is clicked down
}
```

Some of these buttons are straightforward. The letter buttons `A`, `B`, `X`, and `Y` correspond to the face buttons on the right side of the controller with the same letter. The `Back` and `Start` buttons are to the left and right of the guide buttons in the center of the controller.

The `LeftShoulder` and `RightShoulder` buttons are the thin buttons on the top left and right of the gamepad. The `LeftStick` and `RightStick` are buttons that set when a player depresses the left and right stick into the controller. The `GamePadButtons` structure also contains a property called `BigButton`. This button is available on a different type of Xbox 360 gamepad called the Big Button controller. We talk about the Big Button controller along with other types of controllers later in this chapter.

Reading Gamepad Direction Pad

Along with the face buttons, the Xbox 360 gamepad offers digital input from a directional pad on the left side of the controller. The `GamePad.DPad` property returns a `GamePadDPad` structure, which contains `ButtonState` properties for the `Up`, `Down`, `Left`, and `Right` directions. Table 12.6 contains a list properties exposed by the `GamePadDPad` structure.

Table 12.6 **Properties of GamePadDPad**

Property	Type	Description
`Up`	`ButtonState`	Pressed state of the Up DPad direction
`Down`	`ButtonState`	Pressed state of the Down DPad direction
`Left`	`ButtonState`	Pressed state of the Left DPad direction
`Right`	`ButtonState`	Pressed state of the Right DPad direction

To determine whether the directions are pressed, use the following lines of code:

```
// Determine if DPad directions are being pressed
if (currentGamePadState.DPad.Left == ButtonState.Pressed)
{
    // Left DPad button is pressed
}
else if (currentGamePadState.DPad.Right == ButtonState.Pressed)
{
    // Right DPad button is pressed
}
if (currentGamePadState.DPad.Up == ButtonState.Pressed)
{
    // Up DPad button is pressed
}
else if (currentGamePadState.DPad.Down == ButtonState.Pressed)
{
    // Down DPad button is pressed
}
```

It is important to use else if conditions only on directions that are opposite from one another. It is possible to have two directions pressed if they are next to each other. For example, a user can press `Down` and `Right` at the same time.

Reading Gamepad Thumb Sticks

The gamepad has four analog states: two for the left and right thumb sticks and two for the left and right triggers. To read the state of the thumb sticks, use the `GamePadState.GamePadThumbSticks` property. The `GamePadThumbSticks` structure contains two properties: Left and Right both of which return Vector2 values. The Vector2 value represents the thumb stick displacement in the X and Y directions. These floating points range from –1.0 to 1.0. Table 12.7 contains a list of properties exposed by the `GamePadThumbSticks` structure.

Table 12.7 **Properties of GamePadThumbSticks**

Property	Type	Description
`Left`	`Vector2`	Position of the left thumb stick
`Right`	`Vector2`	Position of the right thumb stick

To read the current X and Y position of the left thumb stick, use the following line of code:

```
Vector2 leftThumbStick = currentGamePadState.ThumbSticks.Left;
```

Reading Gamepad Triggers

Finally, the last type of gamepad state are the triggers. To read the current trigger position, use the `GamePadState.Triggers` property. It returns a `GamePadTriggers` structure, which contains two float values in the `Left` and `Right` properties. Each triggers value is from 0.0, which indicates not depressed, to 1.0, which indicates that the trigger is fully depressed. Table 12.8 contains a list of the properties exposed by the `GamePadTriggers` structure.

Table 12.8 **Properties of GamePadTriggers**

Property	Type	Description
Left	float	Position of the left trigger depression
Right	float	Position of the right trigger depression

To read the current position of the right trigger, use the following line of code:

```
float rightTrigger = currentGamePadState.Triggers.Right;
```

Moving Sprites Based on Gamepad Input

Now let's get the sprite moving around on the screen again. This time, you use the gamepad as your input device. Use the left thumb stick to move the sprite around on the screen. The A and X buttons flip the sprite horizontally and vertically. The direction pad's up and down buttons scale the sprite up and down. First, declare member variables to hold the sprite's texture, position, and other values. In your game class, add the following member variables:

```
Texture2D spriteTexture;
Vector2 spritePosition = new Vector2(200, 200);
Vector2 spriteCenter;
float spriteSpeed = 100f;
GamePadState lastGamePadState;
float rotation;
float scale = 1.0f;
SpriteEffects flipOptions;
```

Next, load a texture to use as your sprite and set the sprite's center based on the texture width and height. In the game's `LoadContent` method, add the following lines of code:

```
spriteTexture = Content.Load<Texture2D>("XnaLogo");
spriteCenter.X = spriteTexture.Width / 2.0f;
spriteCenter.Y = spriteTexture.Height / 2.0f;
```

Now, add logic to the game's `Update` method to read the current gamepad state and set the sprite's position, rotation, scale, and flip orientation. In your game's `Update` method, add the following lines of code:

```
// Read the current gamepad state
GamePadState currentGamePadState = GamePad.GetState(PlayerIndex.One);

// Move the sprite based on the left thumb stick
spritePosition.X += spriteSpeed *
                    currentGamePadState.ThumbSticks.Left.X *
                    (float)gameTime.ElapsedGameTime.TotalSeconds;
spritePosition.Y -= spriteSpeed *
                    currentGamePadState.ThumbSticks.Left.Y *
                    (float)gameTime.ElapsedGameTime.TotalSeconds;

// Flip the sprite horizontally if the A button is pressed
if (currentGamePadState.Buttons.A == ButtonState.Pressed &&
    lastGamePadState.Buttons.A == ButtonState.Released)
{
    flipOptions ^= SpriteEffects.FlipHorizontally;
}
// Flip the sprite vertically if the X button is pressed
if (currentGamePadState.Buttons.X == ButtonState.Pressed &&
    lastGamePadState.Buttons.X == ButtonState.Released)
{
    flipOptions ^= SpriteEffects.FlipVertically;
}

// Set rotation based on trigger positions
rotation = (-currentGamePadState.Triggers.Left +
                  currentGamePadState.Triggers.Right) *
                  MathHelper.Pi;

// Set scale with DPad Up and Down buttons
if (currentGamePadState.DPad.Up == ButtonState.Pressed)
    scale += (float)gameTime.ElapsedGameTime.TotalSeconds;
else if (currentGamePadState.DPad.Down == ButtonState.Pressed)
    scale -= (float)gameTime.ElapsedGameTime.TotalSeconds;

// Save the last frames state
lastGamePadState = currentGamePadState;
```

The only remaining bit of code you need now is the sprite's draw code. In your game's `Draw` method, add the following lines of code:

```
// Draw the sprite
spriteBatch.Begin();
spriteBatch.Draw(spriteTexture, spritePosition, null, Color.White,
                 rotation, spriteCenter, 1.0f, flipOptions, 0);
spriteBatch.End();
```

Run the resulting game. The gamepad controls the sprite's position, rotation, scale, and flip orientation (see Figure 12.5).

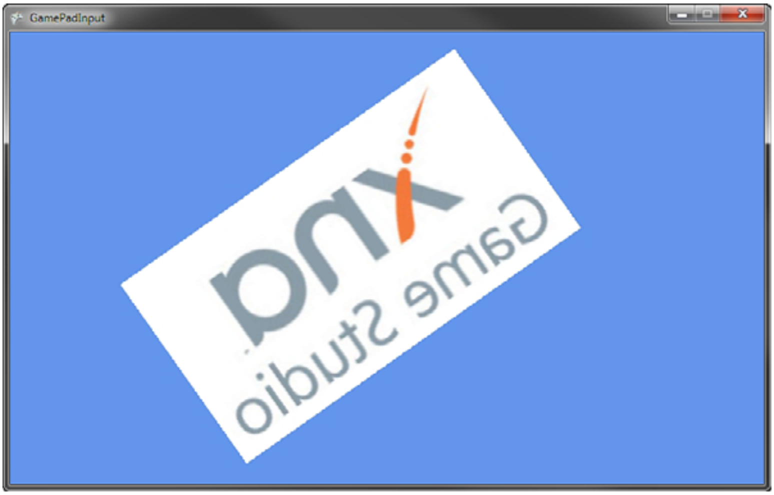


Figure 12.5 Moving, rotating, and scaling sprite using gamepad input

Thumb Stick Dead Zones

When you request the state of the gamepad, by default, a dead zone is applied to the thumb sticks. The thumb sticks on controllers do not always return directly to the center resting position when released. If you use the raw values of the thumb sticks, your game continues to read thumb stick values other than 0. This can cause the game character to move when the user is not pressing the thumb stick. The `GamePadDeadZone` enum defines three types of dead zones. The default used when calling `GetState` is `IndependentAxes`, which compares each axis separately to determine whether the values should be set to 0. The `Circular` dead zone uses the X and Y positions in combination to determine whether the values should be set to 0. The last type is `None`, which causes `GetState` to return the raw values from the thumb sticks. Use `None` to implement your own dead zone processing.

Other Types of Controllers

Up to this point, we have focused on the Xbox 360 controller gamepad, but the XNA Game Studio APIs support a number of different gamepad types. To determine the type of gamepad, use the `GamePad.GetCapabilities` method to return a `GamePadCapabilities` structure.

The type of gamepad is defined by the `GamePadType` property. It returns a `GamePadType` enum value for the many different types of gamepads. Table 12.9 contains the enumeration values for `GamePadType`.

Table 12.9 **GamePadType Enumerations**

Name	Description
`GamePad`	Default Xbox 360 controller
`Guitar`	Guitar controller used for music games
`AlternateGuitar`	Guitar controller used for music games
`DrumKit`	Drum controller used for music games
`ArcadeStick`	Arcade stick used for fighting or arcade games
`DancePad`	Dance pad used for rhythmic dancing games
`FlightStick`	Flight stick used for flying games
`Wheel`	Wheel controller used for driving games
`BigButtonPad`	Big Button controller used for trivia games
`Unknown`	The controller type is unknown and not supported

Using the `GamePadType`, you can determine whether the gamepad is the default controller or whether another type of gamepad is connected. This is helpful for games that require specific controllers, such as the `DancePad` or `DrumKit`. For racing games and flying games, the `Wheel` and `FlightStick` gamepad types can be used by a player. Your game logic is most likely that you want to handle user input differently from a Wheel than an Xbox 360 controller. To determine whether player two's gamepad is a `Guiter`, use the following lines of code:

```
GamePadCapabilities gamePadCapabilities =
GamePad.GetCapabilities(PlayerIndex.Two);
if (gamePadCapabilities.GamePadType == GamePadType.Guitar)
{
    // Player two is using a Guitar
}
```

> **Note**
>
> A `GamePadType` is `Unknown` when the gamepad type is not supported by XNA Game Studio.

Along with the `GamePadType`, the `GamePadCapabilities` contains many properties that enable you to determine whether the controller supports specific buttons, thumb sticks, or triggers.

The big button controller has a unique `BigButton` along with four smaller `A`, `B`, `X`, `Y` buttons (see Figure 12.6). This controller is great for trivia games where users want to quickly hit a big button to ring in their answer. The four other buttons can then be used to specify their answers.

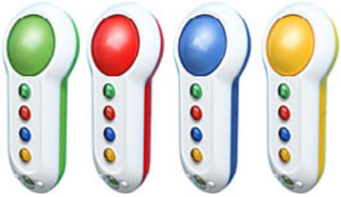


Figure 12.6 Big Button controllers

Is the Gamepad Connected?

What happens if you read the state of a gamepad that is not connected? Although XNA Game Studio supports up to four `PlayerIndex` values when calling `GamePad.GetState`, it does not mean that four gamepads are connected. To determine whether a gamepad is connected, use the `GamePadState.IsConnected` property. This is a simple boolean value that tells you whether the gamepad is connected or not. To determine whether player four's controller is connected, use the following lines of code:

```
GamePadState currentGamePadState = GamePad.GetState(PlayerIndex.Four);
bool playerFourConnected = currentGamePadState.IsConnected;
```

> **Note**
>
> The `GamePadCapabilities` structure also contains a `IsConnected` property that can be used to determine whether the gamepad is connected.

Multitouch Input For Windows Phones

The final type of input device supported by XNA Game Studio APIs are multitouch displays available on all Windows Phone devices. The multitouch APIs are supported on all Windows Phones.

Multitouch devices are unique in that the user can directly touch the display screen and seem to be actually touching and interacting with the images on the screen. This provides an intimate level of interaction that is common on hand-held devices.

> **Note**
>
> XNA Game Studio 4 is not the first version of XNA Game Studio to support multitouch. The Microsoft XNA Game Studio 3.1 Zune Extensions provides similar APIs for Zune HD devices.

Multitouch devices are represented in XNA Game Studio using the `TouchPanel` class type. There are two ways you can read multitouch data from the `TouchPanel`.

The first is to poll the current state of the TouchPanel using the `GetState` method. This returns a number of touch locations where the user has made contact with the multitouch display. All Windows Phone devices support at least four touch locations at a minimum. After you determine the current touch locations, you can use that information to move, rotate, or scale graphics on the display.

The second way to read multitouch data is to utilize gestures. A *gesture* is an action that is performed by the user that can occur over time and must be processed to determine which gesture the user intended to use. For example, the user can drag his finger across the screen.

Reading the `TouchPanel` Device State

Just like the previous input device APIs, the `TouchPanel` provides a state-based API that allows the current state of the multitouch device to be polled at regular intervals. This state can then be used to perform in game actions. Use the `TouchPanel` class to read the current state of a multitouch device. The `GetState` method returns a `TouchCollection` that contains all of the current touch locations. Table 12.9 and 12.10 contain a list of methods and properties for the `TouchPanel` object.

Table 12.9 **Methods of TouchPanel**

Method	Description
`GetState`	Returns collection of current touch locations
`GetCapabilities`	Returns the current touch panel capabilities
`ReadGesture`	Returns the next available gesture

Table 12.10 **Properties of TouchPanel Continued**

Property	Type	Description
`DisplayWidth`	`int`	The width of the touch panel
`DisplayHeight`	`int`	The height of the touch panel
`DisplayOrientation`	`DisplayOrientation`	Orientation of the touch panel
`EnabledGestures`	`GestureType`	Gestures enabled on the touch panel
`IsGestureAvailable`	`bool`	True if a gesture can be read
`WindowHandle`	`IntPtr`	Window to receive touch input

To return the current collection of touch locations, add the following line of code to your game's `Update` method:

```
TouchCollection currentTouchState = TouchPanel.GetState();
```

The `TouchCollection` contains all of the touch locations that are currently pressed or were released since the last frame. To read each `TouchLocation`, you can loop over the `TouchCollection` and read each of the locations properties:

```
// Loop each of the touch locations in the collection
foreach (TouchLocation touchLocation in currentTouchState)
{
    // The uniquie ID of this locaiton
    int touchID = touchLocation.Id;
    Vector2 touchPosition = touchLocation.Position;
    TouchLocationState touchState = touchLocation.State;
}
```

The Position and State can then be used by your game to make interactive game play decisions. Table 12.11 contains a list of the properties exposed by the `TouchLocation` structure.

Table 12.11 **Properties of TouchLocation**

Property	Type	Description
`Position`	`Vector2`	The position of the touch location
`State`	`TouchLocationState`	The current state of the touch location
`Id`	`int`	The unique ID of the touch location

Each touch location contains three properties. Use the `Position` property to determine where on the screen the `TouchLocation` is located. Each `TouchLocation` can be in any of four states. `Pressed` means this specific `TouchLocation` was just pressed by the user. `Moved` means that this specific `TouchLocation` was already tracked and that it has moved. `Released` mean that the `TouchLocation` that was being tracked is no longer pressed by the user, and he or she released his or her finger. Invalid means the `TouchLocation` is invalid and should not be used. Use the `TouchLocation.State` property to determine the current state of a returned location. The final property `Id` is used to give each `TouchLocation` a unique ID that can be used to determine whether two locations across `GetState` calls are the same point. The index of the touch location in the `TouchCollection` should not be used as the identifier because the user can pick up a finger but leave others pressed.

> **Note**
>
> If a user touches the screen and then his or her finger slides off the screen, the `TouchLocation` for that finger changes state to `Released`.

Determine Number of Touch Points

The `TouchPanel` class contains the `TouchPanel.GetCapabilities` method and returns `TouchPanelCapabilities` a structure. Table 12.12 contains the properties exposed by the `TouchPanelCapabilities` structure.

Table 12.12 **Properties of TouchPanelCapabilities**

Property	Type	Description
`IsConnected`	`Vector2`	Returns true if a multitouch capable device is connected
`MaximumTouchCount`	`int`	Returns the maximum number of `TouchLocation`s that can be tracked by the `TouchPanel`

To determine the number of touch points the device supports, use the following lines of code:

```
// Query the capabilities of the device
TouchPanelCapabilities touchCaps = TouchPanel.GetCapabilities();
// Check to see if a device is connected
if(touchCaps.IsConnected)
{
    // The max number of TouchLocation points that can be tracked
    int maxPoints = touchCaps.MaximumTouchCount;
}
```

After you have determined that a touch device is connected, use the `MaximumTouchCount` property to read how many `TouchLocation` points that the `TouchPanel` can track at a time.

> **Note**
>
> Windows Phone games should never require more than four touch points to play the game. More than four can be used if they are available, but they should not be required to play the game successfully.

TouchPanel Width, Height, and Orientation

By default, the width, height, and orientation are set for the `TouchPanel`. If your game supports autorotation, the orientation of the touch panel is automatically updated to match the graphics. `TouchPanel` does provide three properties to read their current values but also enables you to override their values. In most cases, this is not useful, but it can be useful if you require your input in a different scale than the display resolution. If your display is 480 wide and 800 tall but you want `TouchLocation` positions that range from 400 wide to 500 tall, you can use the following lines of code:

```
TouchPanel.DisplayWidth = 400;
TouchPanel.DisplayHeight = 500;
```

Moving Sprite Based on Multitouch Input

It is time to get the sprite moving around the screen again. This time, you use your newly learned multitouch APIs. Use the first touch location to move the sprite around the screen, and use the second touch location to rotate and scale the sprite. First, you need member variables to store the sprite texture, position, center, rotation, and scale. Add the following lines of code to your `Game` class:

```
Texture2D spriteTexture;
Vector2 spritePosition;
Vector2 spriteCenter;
float rotation = 0;
float scale = 1;
```

Next, load a texture to use as your sprite and set the sprite's center based on the texture width and height. In the game's `LoadContent` method, add the following lines of code:

```
spriteTexture = Content.Load<Texture2D>("XnaLogo");
spriteCenter.X = spriteTexture.Width / 2.0f;
spriteCenter.Y = spriteTexture.Height / 2.0f;
```

Now, read the current state of the `TouchPanel` and update your sprite. Add the following lines of code to your game's `Update` method:

```
// Get current touch state
TouchCollection currentTouchState = TouchPanel.GetState();
// Loop over each touch point
for (int i = 0; i < currentTouchState.Count; i++)
{
    TouchLocation currentLocation = currentTouchState[i];
    // Only process points that were just pressed or are moving
    if (currentLocation.State == TouchLocationState.Pressed ||
        currentLocation.State == TouchLocationState.Moved)
    {
        // Use the 1st point for the sprite location
        if (i == 0)
        {
            spritePosition = currentLocation.Position;
        }
        // Use the 2nd point for rotation and scale
        if (i == 0)
        {
            // Check distance between the 1st and 2nd points
            Vector2 positionToSprite = currentLocation.Position - spritePosition;
            float distance = positionToSprite.Length();
            scale = distance / (spriteTexture.Width / 2);
            // Normalize to get direction vector
            positionToSprite.Normalize();
            // Use the direction between fingers to determine rotation angle
            float cosAngle = Vector2.Dot(positionToSprite, Vector2.UnitX);
            if (spritePosition.Y > currentLocation.Position.Y)
                rotation = 2 * MathHelper.Pi - (float)Math.Acos(cosAngle);
            else
                rotation = (float)Math.Acos(cosAngle);
        }
        else
            break;
    }
}
```

In the previous code, notice that you loop over the first two touch locations. Each location is used only if its state is set to `Pressed` or `Moved`. `Released` and `Invalid` points are not used. The first touch point is used to set the sprite's position on the screen. The second point is then used to determine the amount to scale and rotate the sprite. To determine the scale amount, create a vector from the second touch location to the first touch location. The distance of this vector is then calculated. The final scale divides this distance by the width of the texture to achieve the proper scale value. To determine the rotation of the sprite, find the angle of the line formed between the first and second finger and the unit X vector. You can then find the cosine of the angle between these two vectors by using the vector dot product. The vector dot product is equal to the cosine of the angle between them multiplied by the length of each vector. To simplify, make sure both vectors have a length of one also known as unit length. The `Vector2.UnitX` vector already has a length of one, so you don't need to normalize it. The vector formed by the two touch locations is normalized. This sets the vector's length to one and creates a direction vector. Now that both vectors have a length of one, the cosine of the angle between them can be found by just using the vector dot product. To get the final angle, take the arccosine of the dot product. This gives you the angle between the vector formed by the two touch locations and the unit X vector in radians.

Finally, draw the sprite to the screen using the values you determined. To draw the sprite to the screen, use the following lines of code in your game's `Draw` method:

```
// Draw the sprite
spriteBatch.Begin();
spriteBatch.Draw(spriteTexture, spritePosition, null, Color.White,
                    rotation, spriteCenter, scale, SpriteEffects.None, 0);
spriteBatch.End();
```

Reading Gestures from the `TouchPanel`

Gestures provide a richer and higher level abstraction than the raw state data you can access by reading the `TouchPanel` state. Gestures are performed by the user over time and require that the system continuously process user input to determine whether an enabled gesture has been performed. `TouchPanel` provides the capability to define which gestures should be tracked by using the `EnabledGestures` property and the `GestureType` enumeration flags. By default, the `TouchPanel` has no gestures enabled because gesture processing requires additional CPU processing. You should limit which gestures you enable to the minimum you currently need to keep the processing cost to a minimum.

Table 12.13 contains the enumeration values for `GamePadType`.

Table 12.13 **GestureType Enumeration**

Name	Description
`None`	No gestures should be processed.
`Tap`	A short touch at a single point on the screen.
`DoubleTap`	Two repeated taps on the screen near a single point.
`Hold`	One second touch at a single point on the screen.
`HorizontalDrag`	A touch that moves horizontally across the screen without lifting the finger.
`VerticalDrag`	A touch that moves vertically across the screen without lifting the finger.
`FreeDrag`	A touch that moves across the screen without lifting the finger.
`Pinch`	Two touch points that are pull closer or further away in a pinching motion.
`Flick`	A touch that is quickly moved in a flicking motion across the screen.
`DragComplete`	Used to signal when one of the drag gestures is completed.
`PinchComplete`	Used to signal when the pinch gesture is completed.

Enabling the gestures you want to respond to is a simple as combining multiple `GestureType` flags using the `TouchPanel.EnabledGestures` static property. The following lines of code enables two gestures:

```
TouchPanel.EnabledGestures = GestureType.Tap | GestureType.FreeDrag;
```

The gesture system then continuously checks for the tap and free drag gestures while the game runs. In our game's input logic code, you then need to check to see whether any gestures have been detected. This is done by calling the `TouchPanel.IsGestureAvailable` static property. This property returns true at least one gesture is ready to be processed. The `TouchPanel.ReadGesture` method is then used to read the next gesture in the queue and remove it from the queue.

The following code reads all of the available gestures from the `TouchPanel`.

```
// Loop while there are still gestures to read
while (TouchPanel.IsGestureAvailable)
{
    // Read the next gesture
    GestureSample gesture = TouchPanel.ReadGesture();

    // Use a switch statement to detemine which type of gesture has been recieved
    switch (gesture.GestureType)
    {
        case GestureType.Tap:
            // Process tap gesture
            break;
        case GestureType.FreeDrag:
            // Process free drag gesture
            break;
    }
}
```

After each `GestureSample` is read, the game can use the gesture input to take some type of action. A `GestureSample` is a structure that holds all of the data relevant to a gesture. Using a switch statement is helpful to determine which type of gesture is used and to take different game actions based on the gesture.

Table 12.14 contains the properties exposed by the `GestureSample` structure.

Table 12.14 **Properties of GestureSample Continued**

Property	Type	Description
GestureType	GestureType	The type of gesture that occurred
Position	Vector2	First touch location of the gesture
Position2	Vector2	Second touch location of the gesture used in multiple point gestures like Pinch

Table 12.14 **Properties of GestureSample Continued**

Property	Type	Description
Delta	Vector2	Delta information about the gesture such as the movement in a FreeDrag gesture
Delta2	Vector2	Second delta information used in multiple point gestures
Timestamp	Timespan	Span of time in which the gesture occurred

Depending on the type of gesture, the different property values of the `GestureSample` can be used when taking action within your game. For example, if you use the `Flick` gesture in your game to scroll the map, you might care about only the `Delta` property so you would know which direction the flick occurred in and with how much velocity. In other situations, you might care about the position of the gesture when using the `DoubleTap` gesture for selection. In that case, you would use the `Position` property.

Displaying GestureSample Data

Now that you have learned how to enable and read gestures let's create a simple application that displays the last five gestures and all of their corresponding `GestureSample` properties.

First, you will need a `List` to store all of the processed gestures in so you can print them later when drawing. Add the following member variable to your game:

```
List<GestureSample> gestures = new List<GestureSample>();
```

Next, in your games `Update` method, add the following lines of code:

```
// Loop while there are still gestures to read
while (TouchPanel.IsGestureAvailable)
{
    // Add each to the stack
    gestures.Add(TouchPanel.ReadGesture());
}
```

This continuously reads all of the available gestures and adds them to the `List` for later processing.

Now in your games `Draw` method, add the following lines of code to loop over all of the gestures and write their values.

```
spriteBatch.Begin();
Vector2 textPosition = new Vector2(5, 5);

// Loop over all of the gestures and draw their information to the screen
for (int i = gestures.Count-1; i >= 0; i–)
{
    // Draw all of the properties of the gesture
    GestureSample gesture = gestures[i];
    spriteBatch.DrawString(spriteFont, "GestureType: " +
                             gesture.GestureType.ToString(), textPosition,
Color.White);
    textPosition.Y += 25;
    spriteBatch.DrawString(spriteFont, "Position: " +
                            gesture.Position.ToString(), textPosition, Color.White);
    textPosition.Y += 25;
    spriteBatch.DrawString(spriteFont, "Position2: " +
                            gesture.Position2.ToString(), textPosition,
                            Color.White);
    textPosition.Y += 25;
    spriteBatch.DrawString(spriteFont, "Delta: " +
                             gesture.Delta.ToString(), textPosition, Color.White);
    textPosition.Y += 25;
    spriteBatch.DrawString(spriteFont, "Delta2: " +
                             gesture.Delta2.ToString(), textPosition, Color.White);
    textPosition.Y += 25;
    spriteBatch.DrawString(spriteFont, "Timestamp: " +
                             gesture.Timestamp.ToString(), textPosition,
                             Color.White);
    textPosition.Y += 40;
}
spriteBatch.End();

// Remove any gestures that wont print on the screen
if (gestures.Count > 5)
    gestures.RemoveRange(0, gestures.Count - 5);
```

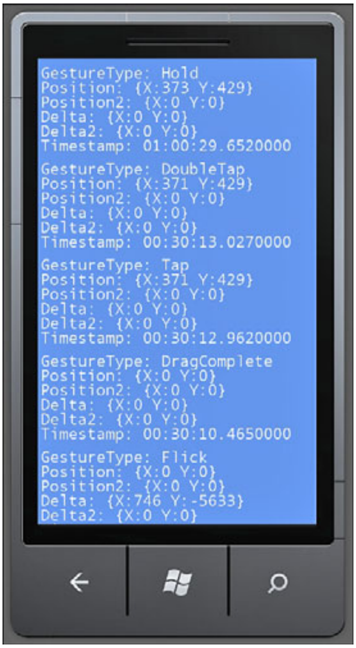
In the previous code, we loop over the `List` of `GestureSample` structures in reverse order and write out each of the properties. We then check the size of the list so that we can remove any gestures that would not be displayed on the screen.

Running the sample shows results similar to those seen in Figure 12.7.

Windows Phone Sensors and Feedback

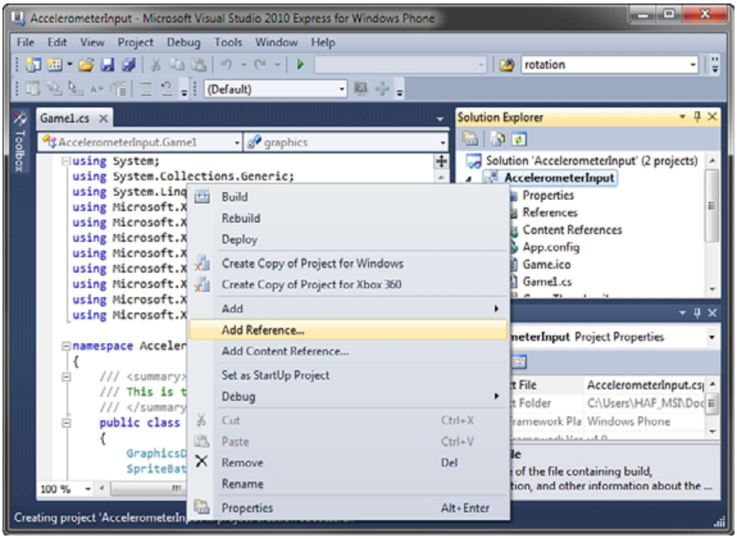
Now we talk about APIs that are specific for Windows Phone devices. These APIs are part of the `Microsoft.Devices.Sensors` and `System.Device.Location` namespaces and require that you add a reference to your project. To use the APIs, perform the following steps for your game:

1. Locate the **Solution Explorer** in Visual Studio.
2. **Right-click** your project and select **Add Reference** (see Figure 12.8).
3. Locate and select the **System.Device** assembly from the list of assemblies under the .Net tab, and press **Add**.



```
GestureType: Hold
Position: {X:373 Y:429}
Position2: {X:0 Y:0}
Delta: {X:0 Y:0}
Delta2: {X:0 Y:0}
Timestamp: 01:00:29.6520000
GestureType: DoubleTap
Position: {X:371 Y:429}
Position2: {X:0 Y:0}
Delta: {X:0 Y:0}
Delta2: {X:0 Y:0}
Timestamp: 00:30:13.0270000
GestureType: Tap
Position: {X:371 Y:429}
Position2: {X:0 Y:0}
Delta: {X:0 Y:0}
Delta2: {X:0 Y:0}
Timestamp: 00:30:12.9620000
GestureType: DragComplete
Position: {X:0 Y:0}
Position2: {X:0 Y:0}
Delta: {X:0 Y:0}
Delta2: {X:0 Y:0}
Timestamp: 00:30:10.4650000
GestureType: Flick
Position: {X:0 Y:0}
Position2: {X:0 Y:0}
Delta: {X:746 Y:-5633}
Delta2: {X:0 Y:0}
```

Figure 12.7 Gesture sample that displays gesture properties


AccelerometerInput - Microsoft Visual Studio 2010 Express for Windows Phone
File
Edit
View
Project
Debug
Tools
Window
Help
rotation
(Default)
Toolbox
Game1.cs
AccelerometerInput.Game1
graphics
Build
Rebuild
Deploy
Create Copy of Project for Windows
Create Copy of Project for Xbox 360
Add
Add Reference...
Add Content Reference...
Set as StartUp Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename
Properties Alt+Enter
Solution Explorer
Solution 'AccelerometerInput' (2 projects)
AccelerometerInput
Properties
References
Content References
App.config
Game.ico
Game1.cs
meterInput Project Properties
t File
AccelerometerInput.csj
t Folder
C:\Users\HAF_MSI\Doc
ramework Pla
Windows Phone
le
of the file containing build,
tion, and other information about the ...
100 %
Creating project 'AccelerometerIn

```
using System;
using System.Collections.Generic;
using System.Linq
using Microsoft.X
using Microsoft.X
using Microsoft.X
using Microsoft.X
using Microsoft.X
using Microsoft.X
using Microsoft.X
using Microsoft.X

namespace Acceler
{
    /// <summary>
    /// This is t
    /// </summary
    public class
    {
        GraphicsD
        SpriteBat
```

Figure 12.8 Project right-click menu displaying the add reference

4. Repeat Steps 1, 2, and 3 to add the ***Microsoft.Devices.Sensors*** assembly.
5. Repeat Steps 1, 2, and 3 to add the ***Microsoft.Phone*** assembly.
6. At the top of code files, include the following lines of code after the existing using statement:

   using Microsoft.Devices.Sensors;

   using System.Device.Location;

Now you are ready to use all of the great sensors and location services that are available on Windows Phone devices. These include the accelerometer to read the acceleration changes that occur to the phone, location services to enable you to determine the phone's position on the planet, and APIs to cause the phone to vibrate.

Because the following APIs are specific to Windows Phone devices, they are not available on all of the other platforms supported by XNA Game Studio. If you build a multiplatform game and want to share code, you need to write the Windows Phone-specific code blocks in `#if WINDOWPHONE` blocks:

```
#if WINDOWS_PHONE
    // Your Windows Phone specific code
#endif
```

Acceleration Data using the Accelerometer

All Windows Phone devices provide access to an accelerometer sensor. An accelerometer is used to measure force changes to the phone device from gravity or by forces exerted by the user. The accelerometer sensor is sensitive enough to determine when the user tilts the device slightly. It also works great to detect when the user shakes or swings the phone.

Reading Acceleration Data

To read acceleration data, you need to construct a new instance of an `Accelerometer`. The `Accelerometer` class is used to start, stop, and read back changes that occur to the accelerometer while it is currently running. Create a member variable to store the `Accelerometer` in your game class:

```
Accelerometer accelerometer;
```

This stores the instance across your game class so you can access the accelerometer in other parts of the game. To create an instance of the `Accelerometer` in your game's `Initialize` method, add the following line of code:

```
accelerometer = new Accelerometer();
```

Unlike the XNA Game Studio input type, the `Accelerometer` uses an event-based model to return new data from the accelerometer. Data is returned by subscribing to the

`Accelerometer.ReadingChanged` event and implementing an event handler method in your game. To subscribe to the `ReadingChanged` event, add the following lines of code after your previous line to create the instance of `Accelerometer`:

```
accelerometer.ReadingChanged += new
EventHandler<AccelerometerReadingEventArgs>(accelerometer_ReadingChanged);
```

Now, implement the `accelerometer_ReadingChanged` method that is called when the accelerometer receives new data. Add the following method to your game class:

```
public void accelerometer_ReadingChanged(object sender,
➥AccelerometerReadingEventArgs e)
{
    Vector3 accelerometerReading;
    accelerometerReading.X = (float)e.X;
    accelerometerReading.Y = (float)e.Y;
    accelerometerReading.Z = (float)e.Z;
    // Update velocity
    spriteVelocity.X += accelerometerReading.X * spriteSpeed;
    spriteVelocity.Y += accelerometerReading.Y * spriteSpeed;
}
```

The reading change event converts the `AccelerometerReadingEventArgs` parameter to something more useful for your game, which is a `Vector3`. In the previous case, you just used a local variable. In your game, you want to store this data or change another variable over successive calls.

To have the accelerometer start reading data, you must call the `Start` method. In the example, we start the accelerometer right away in the game's `Initialize` method. For your game, you should not start to read accelerometer data until it is needed. For example, you don't need to start the accelerometer when the game starts unless you use the accelerometer in your menus. Wait until your level starts before starting the accelerometer. Add the following lines of code after you set the `ReadingChanged` in your game's `Initialize` method:

```
try
{
    accelerometer.Start();
}
catch (AccelerometerFailedException e)
{
    // Accelerometer not supported on this platform
    // We are going to exit.
    Exit();
}
```

The `Start` method throws a `AccelerometerFailedException` if the accelerometer is not supported on the platform or if you have already called the `Start` method. In the previous code, we exit the example game. In your game, you should determine what the best course of action is.

Now that the accelerometer has started, your game should start to revive reading changed events whenever the phone's accelerometer detects changes.

When your game is not using the accelerometer, you should call the `Stop` method to help save battery life and to help your game's performance. For our example, we use the `Game` classes `OnExiting` method to put the call to the `Stop` method. The default game template does not create an override for `OnExiting`, so you need to add the overridden method to your game class. In your game class, add the following method:

```
protected override void OnExiting(object sender, EventArgs args)
{
    try
    {
        accelerometer.Stop();
    }
    catch (AccelerometerFailedException e)
    {
        // Do nothing since we are already exiting
    }

    base.OnExiting(sender, args);
}
```

Similar to the `Start` method, the `Stop` method can also throw an exception. The `AccelerometerFailedException` is thrown when the accelerometer fails to stop or if it is already stopped.

Moving a Sprite Based on Accelerometer Data

Now that you have the basics of getting accelerometer data, let's do something more fun and exciting. Like the previous examples, you move a sprite based on the user input, but in this case use the accelerometer data.

To build on the previous code example, you need some member variables to hold the sprite texture, position, and velocity. Add the following member variables to your game's class:

```
Texture2D spriteTexture;
Vector2 spritePosition;
Vector2 spriteVelocity;
float spriteSpeed = 50.0f;
int screenWidth = 480;
int screenHeight = 800;
```

Next, load a texture to use as your sprite. In the game's `LoadContent` method, add the following line of code:

```
spriteTexture = Content.Load<Texture2D>("XnaLogo");
```

Next, update the sprite's velocity based on the returned acceleration data. To do this, update the `accelerometer_ReadingChanged` method you implemented previously. Add the following lines of code to the method:

```
// Update velocity
spriteVelocity.X += accelerometerReading.X * spriteSpeed;
spriteVelocity.Y += accelerometerReading.Y * spriteSpeed;
```

The sprite's velocity is now updated based on the accelerometer reading changes and then scaled.

Now, create the sprite update logic that changes the sprite position based on the current velocity. In your game's `Update` method, add the following lines of code:

```
// Update sprite position
spritePosition += spriteVelocity * (float)gameTime.ElapsedGameTime.TotalSeconds;

// Dampen velocity
spriteVelocity *= 1 - (float)gameTime.ElapsedGameTime.TotalSeconds;

// Check sprite bounds
if (spritePosition.X < 0)
{
    spritePosition.X = 0;
    spriteVelocity.X = 0;
}
else if (spritePosition.X > screenWidth - spriteTexture.Width)
{
    spritePosition.X = screenWidth - spriteTexture.Width;
    spriteVelocity.X = 0;
}
if (spritePosition.Y < 0)
{
    spritePosition.Y = 0;
    spriteVelocity.Y = 0;
}
else if (spritePosition.Y > screenHeight - spriteTexture.Height)
{
    spritePosition.Y = screenHeight - spriteTexture.Width;
    spriteVelocity.Y = 0;
}
```

After you update the sprite's position, you dampen the velocity so it slows over time. Then, check the bounds of the sprite so it can't travel off the screen. Otherwise, it permanently flies off the screen. When the sprite hits one of the walls, set the velocity for that direction to 0 to enable the sprite to move easily off the wall.

The final step is to draw the sprite. To do this, add the following lines of code to your game's `Draw` method:

```
// Draw the sprite
spriteBatch.Begin();
spriteBatch.Draw(spriteTexture, spritePosition, Color.White);
spriteBatch.End();
```

Autorotation and Acceleration Data

XNA Game Studio games on Windows Phone support autorotation that enable you as a developer to select which orientations work best for your game. Autorotation can cause a couple of challenges when creating a game that uses the accelerometer.

The first issue is that if your game supports multiple orientations and you expect the user to move and tilt the phone, it could cause the orientation of the game to switch when the user does not intend to change orientations. It is problematic if the screen rotates in the middle of playing a flight simulator evading incoming missiles. To address this concern, you can limit your supported orientations and test that the accelerometer actions you expect your users to use don't trigger the orientations during normal game play. If you find that the orientations are triggered, you can set the supported orientations to just a single orientation and enable users to change the orientation using a menu option in your game.

The other issue is that when your game changes orientations, the acceleration data does not change. If your game supports multiple orientations, your game needs to take into account the current orientation and how that maps to the accelerometers X, Y, and Z coordinates.

Locating a Windows Phone with the Location Service

The Windows Phone provides the capability to enable you to take your games on the go. Games are no longer just played at your home on your Xbox 360 or Windows PC. Because phones are so portable, the phone could be anywhere in the world. This brings some new and interesting opportunities for games. Having a portable and connected game device enables you to design games that take the current location into account. You could also use it to find other players of the game who are nearby. The accuracy of the location information while affected by several environmental factors can be quite accurate within a few meters.

Reading Location Data

Use the `GeoCoordinateWatcher` class to report the phone's current location. You need an instance of the geo coordinate watcher along with the latest coordinate received and the current status. Add the following variables to your game's class:

```
GeoCoordinateWatcher geoWatcher;
GeoCoordinate currentCoordinate;
GeoPositionStatus geoStatus;
```

Similar to acceleration data, the geo coordinate data is returned though an event-based model from `GeoCoordinateWatcher`. You subscribe to two events. Use the

`StatusChanged` to tell you the current state of the location services. After you start the location service, the status can change and this event gives you information about the state the service is currently in. The `PositionChanged` returns when the location service detects that the phone moved to a new location. Create an instance of the `GeoCoordinateWatcher` and subscribe to these events. Add the following lines of code to your game's `Initialize` method:

```
geoWatcher = new GeoCoordinateWatcher(GeoPositionAccuracy.High);
geoWatcher.StatusChanged += new
EventHandler<GeoPositionStatusChangedEventArgs>(geoWatcher_StatusChanged);
geoWatcher.PositionChanged += new
EventHandler<GeoPositionChangedEventArgs<GeoCoordinate>>
➥(geoWatcher_PositionChanged);
geoWatcher.MovementThreshold = 5;
geoWatcher.Start();
```

When you create the instance of `GeoCoordinateWatcher`, you can pass in a `GeoPositionAccuracy` enum value of `High` or `Low`. This tells the geo coordinate watcher which accuracy should be used when returning position changed locations. The `MovementThreshold` property is in meters and refers to the distance the phone needs to travel before the position changed event is fired. Setting this value too low can result in many position changed events caused by noise in the determined location of the device. Finally, call `Start` to start the geo coordinate watcher. Just like the accelerometer, you should call `Start` only after you are ready to receive data because it lowers the battery life of the device. You should also call `Stop` whenever you are finished tracking the user's location.

Now that the `StatusChanged` and `PositionChanged` events are set, implement the `geoWatcher_StatusChanged` and `geoWatcher_PositionChanged` methods to handle these events. Add the following methods to your game class:

```
void geoWatcher_StatusChanged(object sender, GeoPositionStatusChangedEventArgs e)
{
    // Check the current status
    geoStatus = e.Status;
    switch (geoStatus)
    {
        case GeoPositionStatus.Initializing:
            // The location service is still starting up
            break;
        case GeoPositionStatus.Ready:
            // The location service is running and returning location data
            break;
        case GeoPositionStatus.NoData:
            // The locaiton has started but can not current find its location
            break;
        case GeoPositionStatus.Disabled:
            if (geoWatcher.Permission == GeoPositionPermission.Denied)
            {
                // The user has turned off location services
            }
            else
            {
                // The location service failed to start
            }
            break;
    }

}

void geoWatcher_PositionChanged(object sender,
                                GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    // Store the current coordinate
    currentCoordinate = e.Position.Location;
}
```

Now, store the current location coordinate each time the location is updated. `GeoCoordinate` provides number properties that contain data about the phone's current location. The `Latitude` and `Longitude` properties can be used to return their respective positions on the Earth. The `Altitude` property returns the current altitude in meters. The `Course` property returns the current direction the phone is moving in. The property returns a double value between 0 and 360 degrees. The `Speed` property returns the current speed of the phone in the course direction. The final two properties, `HorizontalAccuracy` and `VerticalAccuracy`, return accuracy of the current coordinate. You can use these values to determine whether you want to use a given point depending on the accuracy needs of your application. To display these properties, add the following code to your game's `Draw` method:

```
string locationInfo = "Status: " + geoStatus.ToString() + "\n";
if (currentCoordinate != null)
{
    locationInfo += "Latitude: " + currentCoordinate.Latitude.ToString
➥("0.000") + "\n";
    locationInfo += "Longitude: " +
➥currentCoordinate.Longitude.ToString("0.000") + "\n";
    locationInfo += "Altitude: " + currentCoordinate.Altitude.ToString
➥("0.000") + "\n";
    locationInfo += "Course: " + currentCoordinate.Course.ToString("0.000") + "\n";
    locationInfo += "Speed: " + currentCoordinate.Speed.ToString("0.000") + "\n";
    locationInfo += "HorizontalAccuracy: " +
                    currentCoordinate.HorizontalAccuracy.ToString("0.000") + "\n";
    locationInfo += "VerticalAccuracy: " +
                    currentCoordinate.VerticalAccuracy.ToString("0.000") + "\n";
}
```

```
spriteBatch.Begin();
spriteBatch.DrawString(spriteFont, locationInfo, new Vector2(10, 50),
➥Color.White);
spriteBatch.End();
```

The previous code requires that you have already added a `SpriteFont` to your project and it is loaded into the `spriteFont` variable.

Providing User Feedback using Vibration

The Xbox 360 gamepad is not the only device that supports force feedback using vibration. The Windows Phone provides the capability to start the vibration motor inside the Windows Phone device. The `VibrateController` type provides both a `Start` and `Stop` method for controlling the vibration of the device. The `Start` method takes in a `TimeSpan` for the duration you want the device to vibrate. Although the vibration stops after the given `TimeSpan`, you can also call the `Stop` method to stop the vibration before the duration has expired. To add vibration to your game, you first need an instance of the `VibrateController`. You can obtain an instance by using the `VibrateController.Default` property. Add the following member variable to your game's class:

```
VibrateController vibrateController;
```

In your game's `Initialize` method, add the following line of code:

```
vibrateController = VibrateController.Default;
```

Now all that is left is to start the vibration motors on the device. Use the following line of code to have the device vibrate for three seconds:

```
vibrateController.Start(TimeSpan.FromSeconds(3));
```

If you need to stop the vibration that you have already started, call the `Stop` method using the following code:

```
vibrateController.Stop();
```

Summary

As you can learned from this chapter, there are many types of input devices and data available to your game especially if you are creating a game for a Windows Phone device. We covered how to read input from all of the input devices supported by XNA Game Studio from keyboards and mice to all of the different types of gamepads available for the Xbox 360 and Windows PCs. We also covered the new multitouch APIs available for Windows Phone devices. Finally, we covered the Windows Phone specific sensors that are available for both XNA Game Studio games and Silverlight applications on Windows Phone devices.

This page intentionally left blank

13

Turn Up the Volume

Let's turn up the volume and add some volume to your game. There are multiple ways in which a gamer can add audio to his game, including playing back prerecorded wave files and generating audio dynamically while the game is running. Audio is an essential part of your game and can even play a part in your title's game play.

In this chapter you learn:

- Loading and playback of sound effects
- Adjusting sound effect pitch, pan, and volume
- Microsoft Cross-Platform Audio Creation Tool (XACT)
- Recording audio using a microphone
- Generating dynamic audio

Playing Sound Effects

Now that your game looks stunning with all of the new fancy graphics techniques you learned, it is time to round out your game with some sound effects and music. Notice that although graphics make your game look nice, something is missing and as a player of the game you are not quite as engaged as you know you can be. Sounds play an integral part in our lives and as such can play a huge role in how gamers perceive a game.

As you learn in this chapter, adding sound effects and music to your game is simple using XNA Game Studio. Determining when and where to use the right sound effects and music is not as easy, and time and care should be put into the sound effect and music selection in your game. Often, those who are new to creating games overlook the sound design of a game. What you find is that all of the effort you put into making the game sound great is paid back in immersion and emotion the game provides.

XNA Game Studio provides the capability to play back sounds for a number of different sources: sound effects and music saved to files, sounds recorded through a microphone, and sounds generated dynamically through programming. In this chapter, we discuss each of these along with a tool called the Microsoft Cross-Platform Audio Creation Tool (XACT).

Using SoundEffect for Audio Playback

The simplest way to play audio is to use the `SoundEffect` class. A `SoundEffect` contains the shared resources needed to play audio such as the wave data that makes up the sound. Load the `SoundEffect` by using the content pipeline similar to how you load a texture or other resources you have seen so far.

`SoundEffect` provides two mechanisms when playing audio. The first is fire and forget, which provides an easy-to-use API to play an audio file. The second is instanced-based playback where you request a `SoundEffectInstance` that enables you as the developer to have more control over the playback of the audio file.

Loading from a File

The first step to play an audio file is to add the file to your content project for your game. Right-click your content project and select Add Existing Item. Then, navigate to an existing audio file. XNA Game Studio supports audio files in WAV, WMA, and MP3 formats.

The first bit of code you need is a member variable that holds the `SoundEffect` that you load. Add the following lines of code as a member variable to your game:

```
// Store our SoundEffect resource
SoundEffect soundEffect;
```

Now you need to load the `SoundEffect` using the content pipeline. You can do this by calling the `Load` method on the `ContentManager` of the `Game` class using `SoundEffect` as the generic parameter and the asset name as the only parameter:

```
// Load the SoundEffect resource
soundEffect = Content.Load<SoundEffect>("beep");
```

Fire and Forget Audio Playback

Now that you have loaded the resources for the audio file, you can use the first mechanism for sound playback called fire and forget. It is called fire and forget because after you call play, you don't need to manage the playback of the audio stream. The audio file plays from the start of the file until the end of the file. Although this limits the control you have as a developer on how the playback occurs, it is simple to use.

To play the `SoundEffect` using fire and forget, call the Play method:

```
soundEffect.Play();
```

The `Play` method returns a boolean value that specifies whether the sound is able to play or not. If `Play` returns true, then the sound plays. If false returns, there are too many sounds currently playing, so it cannot play.

The `Play` method also contains an overload with three parameters to control the volume, pitch, and panning of the playback:

```
soundEffect.Play(1.0f, 0.0f, 0.0f);
```

The first parameter is a float value that determines the volume of the sound. The value can range from 0, which indicates no sound, to 1.0, which means the sound should be full volume with respect to the master volume setting. Use the `MasterVolume` static property of `SoundEffect` to set the master volume setting for all `SoundEffects`. The `MasterVolume` values range from 0 to 1 with 1 as the default value.

The second parameter is a float value that determines the pitch that the sound plays in. The pitch value ranges from –1, which lowers the pitch by one octave, to 1, which raises the pitch by one octave. A value of 0 plays the sound as it is stored in the file.

The third parameter controls the pan of the playback. Panning determines how much of the playback occurs from the left and right speakers. The values rage from –1, which comes from only the left speaker, to 1, which comes from only from the right speaker. A value of 0 plays the sound as it was loaded from the file.

> **Note**
>
> You can save files with the audio playing from only one of the sides. If the file is saved with the audio source already panned to one side or the other, a value of 0 still contains the original panning.

`SoundEffect` also exposes two properties that are useful. The first is the Duration property that returns the length as a `TimeSpan` of the `SoundEffect`. The other is `Name` that returns the name of the `SoundEffect`.

Playback Using SoundEffectInstance

The other method to play back sounds requires an instance of the sound effect. The `SoundEffectInstance` provides the capability to control the playback of a single audio stream from a `SoundEffect`. Create a `SoundEffectInstance` by calling the `CreateInstance` method on the `SoundEffect`. A `SoundEffect` can create many instances. Each instance provides a number of controls that enable more advanced playback scenarios.

`SoundEffectInstance` provides methods to `Play`, `Pause`, `Stop`, and `Resume` playback. The `Play` method takes no parameters and works similar to the `SoundEffect.Play` method. It starts the playback of the `SoundEffectInstance` at the current play location. If the sound is paused, calling `Play` resumes playback from where the sound was paused. The `Resume` method is similar and resumes playback from the current sound location.

The `Pause` method does what its name suggests and pauses playback at the current playback location where it can be started again later by calling the `Play` or `Resume` methods.

The `Stop` method stops the sound playback immediately and sets the playback location to the start of the sound. `Stop` provides an overload that takes a boolean value to specify if the playback should stop immediately. If this value is set to false, the playback stops only after it reaches the end of the sound. This is useful when you want to allow the sound to complete but stop any looping that occurs.

Use the `State` property to determine the current state of a `SoundEffectInstance`. It returns a `SoundState` enumeration that contains the values `Playing`, `Paused`, or `Stopped`.

Looping Audio Playback

When a `SoundEffectInstance` is set to loop, the sound continues to play until it is paused or stopped. After the current sound location reaches the end of the duration, the current position sets back to the start where playback continues causing the sound playback to loop until it is paused or stopped.

To set a `SoundEffectInstance` so it loops, set the `IsLooped` property to true. After the property is set to true, the sound loops when it is played. You can set the `IsLooped` property only before `Play` or `Resume` methods are called on the instance. After `Play` or `Resume` are called on an instance, the `IsLooped` property can't change. Otherwise, an `InvalidOperationException` is raised. To go from playing a specific sound with looping behavior back to not looping another, `SoundEffectInstance` must be created.

Adjusting the Pitch, Pan, and Volume of Playback

The `Pitch`, `Pan`, and `Volume` properties can be used to adjust their values. These work in the same way as the `SoundEffect.Play` overload method, which takes volume, pitch, and pan as arguments. `.Pitch` and `Pan` are float values that must be between –1 and 1, whereas `Volume` must be between 0 and 1.

3D Audio Positioning

In most 3D games and applications, the sounds should come from a source in the game. Jet engine noise should sound like it comes from the direction of the engines, the gun fire noises should come from the direction of an enemy, and car horns should sound like they come from a nearby passing car. We perceive that these sounds come from different positions in the world even though they come from a stationary set of speakers. This simulation occurs because different speakers output different amounts of the sound, giving the impression that the audio source is a specific direction from the player. The volume of the sound can also create the illusion that a sound is close or farther away.

`SoundEffectInstance` allows for an `AudioEmitter` and `AudioListener` by using the `Apply3D` method. The `AudioEmitter` type describes the location, direction, and velocity of the object that is the source of the sound that is played back using the `SoundEffectInstance`. To create a new instance of an `AudioEmitter,` use the default parameterless constructor:

```
AudioEmitter audioEmitter;
```

To set the current position of the `AudioEmitter` using the `Position` property, set the location to the position of the object emitting the sounds. If you use a `Matrix` to store the position and orientation of an object in your game, use the `Translation` property of the matrix to get the position of the object. The `AudioEmitter` also contains properties for the `Forward` and `Up` vectors. Use these vectors to determine the direction that the emitter moves along with the `Velocity` property. These properties allow for the sound coming from the `AudioEmitter` to simulate the Doppler effect.

The Doppler effect, in regards to audio, has to do with the change in an audio emitter's frequency when the position and velocity of the object change in relation to the listener of the audio waves. As an object moves toward the listener, the sound waves compress, causing the frequency to increase from the source sound. After the object passes the listener, the audio waves stretch, causing the frequency to lower.

`AudioEmitter` provides a `DopplerScale` property. Use this property to scale the Doppler calculation. The default value is 1.0. Lowering this value decreases the influence of the Doppler on the final sound output. Increasing the value increases the Doppler effect on the final sound output.

> **Note**
>
> `SoundEffect` also provides two static properties that affect the Doppler calculation. Use `DopplerScale` and `DistanceScale` to change the influence of the Doppler calculation.

The `AudioListener` represents the location of the emitter sound. To create an instance of the `AudioListener`, use the default parameterless constructor:

```
AudioListener audioListener;
```

Similar to the `AudioEmitter`, the `AudioListener` provides properties for the current `Position`, `Forward`, and `Up` vectors, and the `Velocity` the listener moves. These properties are used in conjunction with the emitter to determine the final audio output.

After you create the `AudioEmitter` and `AudioListener`, you can call the `Apply3D` method to update the 3D positioning information used by the `SoundEffectInstance`:

```
soundEffectInstance.Apply3D(audioListener, audioEmitter);
```

Adding SoundEffectInstance to Your Game

Now that you learned how the `SoundEffectInstance` works, you can add more advanced audio playback to your game.

The first thing to add to the game is some local variables to store the `SoundEffect`, `SoundEffectInstance`, `AudioEmitter`, and `AudioListener`:

```
// Store our SoundEffect resource
SoundEffect soundEffect;
// Instance of our SoundEffect to control playback
SoundEffectInstance soundEffectInstance;
// Location of the audio source
AudioEmitter audioEmitter;
// Location of the listener
AudioListener audioListener;
```

Next, load the audio file that is used for the source of audio. In the `LoadContent` method, add the following lines of code:

```
// Load the SoundEffect resource
soundEffect = Content.Load<SoundEffect>("beep");
soundEffectInstance = soundEffect.CreateInstance();
audioEmitter = new AudioEmitter();
audioListener = new AudioListener();
```

You just loaded the sound effect from the file. Then, create a new `SoundEffectInstance` that is used to control the playback. An `AudioEmitter` and `AudioListener` are also created with their default constructors.

In the `Update` method, you control the playback of the `SoundEffectInstance`. The first controls you add are for basic sound playback of playing, pausing, stopping, and resuming:

```
// Play the sound
if (currentKeyboardState.IsKeyDown(Keys.Space) &&
    lastKeyboardState.IsKeyUp(Keys.Space))
    soundEffectInstance.Play();
// Pause the sound
if (currentKeyboardState.IsKeyDown(Keys.P) &&
    lastKeyboardState.IsKeyUp(Keys.P))
    soundEffectInstance.Pause();
// Stop
if (currentKeyboardState.IsKeyDown(Keys.S) &&
    lastKeyboardState.IsKeyUp(Keys.S))
    soundEffectInstance.Stop();
// Resume
if (currentKeyboardState.IsKeyDown(Keys.R) &&
    lastKeyboardState.IsKeyUp(Keys.R))
    soundEffectInstance.Resume();
```

Use the `Space`, `P`, `S`, and `R` keys to control the `Play`, `Pause`, `Stop`, and `Resume` methods, respectively. Check the last keyboard state to ensure you don't get repeat keyboard presses. Otherwise, when users press the key, the `Play` or other methods are called multiple times until users release the key.

Next, use the `L` key to control whether the instance should loop or not. If the `L` key is pressed before the first time, the `SoundEffectInstance` plays, and then the sound loops until it is stopped or paused. As mentioned before, if you set the `IsLooped` property after the instance has already been played, the `InvalidOperationException` is raised. As a developer, you need to determine whether a sound needs to loop before playing the sound. Add the additional lines of code to the `Update` method to add the looping behavior:

```
// Start or stop looping
if (currentKeyboardState.IsKeyDown(Keys.L) &&
    lastKeyboardState.IsKeyUp(Keys.L))
    soundEffectInstance.IsLooped = !soundEffectInstance.IsLooped;
```

Next, add controls to set the `Pitch`, `Pan`, and `Volume` of the `SoundEffectInstance`:

```
// Change Pitch
if (currentKeyboardState.IsKeyDown(Keys.Q))
    soundEffectInstance.Pitch = -1.0f;
```

```
else if (currentKeyboardState.IsKeyDown(Keys.W))
    soundEffectInstance.Pitch = 0.0f;
else if (currentKeyboardState.IsKeyDown(Keys.E))
    soundEffectInstance.Pitch = 1.0f;
```

You can use the `Q` key to lower the pitch to the lowest value of –1.0f and use the `E` key to set the pitch to the highest value of 1.0f. Use the `W` key to reset the pitch to the recorded value from the loaded `SoundEffect` file by setting the pitch to a value of 0:

```
// Change Pan
if (currentKeyboardState.IsKeyDown(Keys.Z))
    soundEffectInstance.Pan = -1.0f;
else if (currentKeyboardState.IsKeyDown(Keys.X))
    soundEffectInstance.Pan = 0.0f;
else if (currentKeyboardState.IsKeyDown(Keys.C))
    soundEffectInstance.Pan = 1.0f;
```

You can use the `Z` key to pan the sound all the way to the left by using a value of –1.0f. Use the `C` key to pan the sound all the way to the right by using a value of 1.0f. Use the `X` key to center the playback to the recorded values from the loaded file:

```
// Change Volume
if (currentKeyboardState.IsKeyDown(Keys.B) &&
    soundEffectInstance.Volume <= 0.99f)
    soundEffectInstance.Volume += 0.01f;
else if (currentKeyboardState.IsKeyDown(Keys.V) &&
    soundEffectInstance.Volume >= 0.01f)
    soundEffectInstance.Volume -= 0.01f;
```

Use the `B` and `V` keys to raise and lower the volume of playback, respectively. Check the value of the `Volume` property to ensure the values stay within the required bounds.

Finally, change the position of the audio source by updating the `AudioEmitter.Position` property and setting its value on the `SoundEffectInstance`. Add the following lines of code inside the `Update` method:

```
// Move audio emitter
Vector3 emitterPos = audioEmitter.Position;
if (currentKeyboardState.IsKeyDown(Keys.Right))
    emitterPos.X += (float)gameTime.ElapsedGameTime.TotalSeconds;
else if (currentKeyboardState.IsKeyDown(Keys.Left))
    emitterPos.X -= (float)gameTime.ElapsedGameTime.TotalSeconds;
if (currentKeyboardState.IsKeyDown(Keys.Up))
    emitterPos.Z -= (float)gameTime.ElapsedGameTime.TotalSeconds;
else if (currentKeyboardState.IsKeyDown(Keys.Down))
    emitterPos.Z += (float)gameTime.ElapsedGameTime.TotalSeconds;
```

```
// Store new position
audioEmitter.Position = emitterPos;
// Set the audio emitter and listener values
soundEffectInstance.Apply3D(audioListener, audioEmitter);
```

You can use the `Right` and `Left` arrow keys to move the audio emitter's position in the `X` coordinate. Use the `Up` and `Down` arrow keys to move the emitter's position along the `Z` coordinate. Update the `SoundEffectInstance` by calling the `Apply3D` method passing in the `AudioListener` and `AudioEmitter`.

> **Note**
>
> Calling `Apply3D` is required anytime you adjust the `AudioListener` and `AudioEmitter`, if you want those changes to affect the audio playback. Just changing the values of `AudioListener` and `AudioEmitter` without calling Apply3D does not change the `SoundEffectInstance` playback even if `Apply3D` has already been called.

Microsoft Cross-Platform Audio Creations Tool (XACT)

The Microsoft Cross-Platform Audio Creations Tool (XACT) is a powerful graphical audio-authoring tool that enables you to manage large numbers of source wave files and to control their playback in the game. Using the XACT graphical interface, you can load existing wave files, group the files, and layer them with effects. The provided XNA Game Studio APIs can trigger their playback.

Although the XACT tool provides several features, we cover the basics of the graphical interface and the XACT APIs to enable you to get started using the tool.

> **Note**
>
> XACT is available only for the Windows and Xbox 360 platforms. The Windows Phone is not supported.

Opening XACT

You can find the XACT tool under the Microsoft XNA Game Studio 4.0 program files menu under the Start menu. Select the Tools: folder, and then click the Microsoft Cross-Platform Audio Creation Tool (XACT) menu item. This launches the XACT tool, which looks like Figure 13.1.

The XACT graphical interface contains many menus, icons, and options. We cover the different areas as they are needed in the following examples.

Creating a New XACT Project

In the XACT tool, select File, then New Project. The New Project Path dialog box displays. Because we use the XACT project in our game's content pipeline, it is often easy to use the content folder of the game to store the XACT project file. Locate the content project folder for your game, and then enter a name for your XACT project and make sure it ends with the .xap extension. Click the Save button.

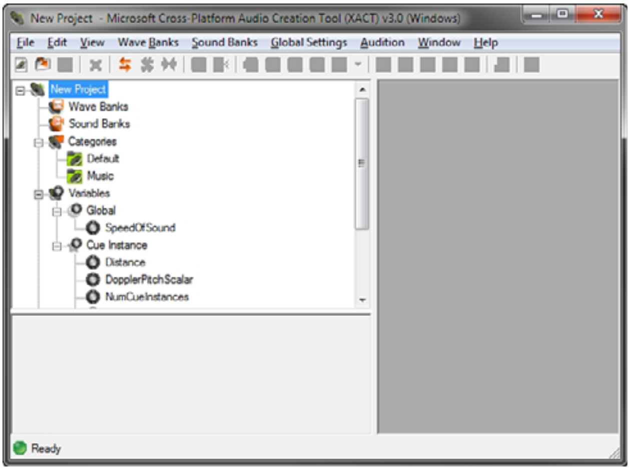

New Project - Microsoft Cross-Platform Audio Creation Tool (XACT) v3.0 (Windows)
File
Edit
View
Wave Banks
Sound Banks
Global Settings
Audition
Window
Help
New Project
Wave Banks
Sound Banks
Categories
Default
Music
Variables
Global
SpeedOfSound
Cue Instance
Distance
DopplerPitchScalar
NumCueInstances
Ready

Figure 13.1 XACT graphical interface

Adding a Wave File to the XACT Project

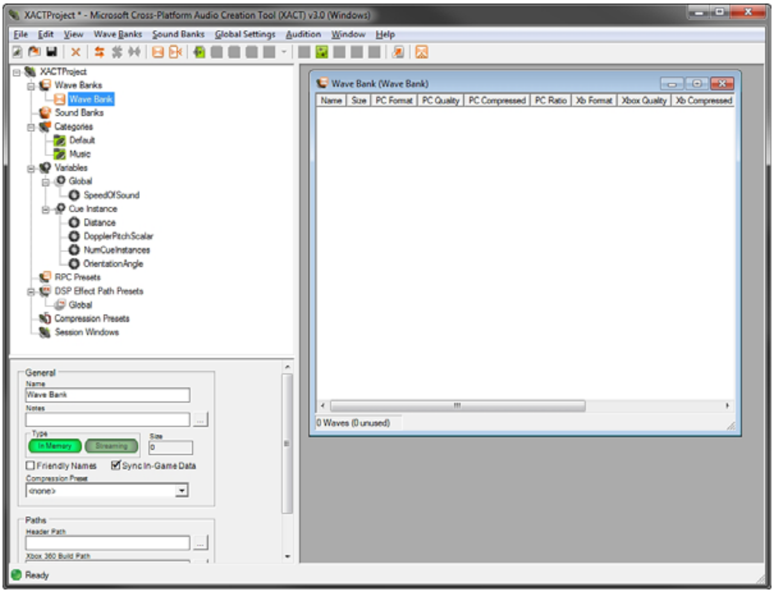
The starting elements of an XACT project are wave files that must be loaded from source recordings. In the following example, you load a single wave file and then play back the sound using the XACT APIs provided in XNA Game Studio.

First, you need a wave bank to load the source wave files into. A wave bank is a collection of wave files that are grouped together. To add a new wave bank, click the Wave Banks menu and select New Wave Bank. A new entry displays under Wave Banks in the XACT project tree with the default name "Wave Bank". You can change the default name by clicking the wave bank in the tree or by right-clicking the wave bank and selecting Rename. A wave bank menu also displays in the right side of the XACT workspace. Your XACT workspace should look like Figure 13.2.

Next, you need to add the first sound bank to the project. A sound bank is a collection of wave banks. To add a new sound bank, click the Sound Banks menu and select New Sound Bank. Just as with the wave bank, the new sound bank is added to the XACT project tree, but this time, the new item displays logically under the Sound Banks tree node. The new sound bank has a default name of Sound Bank, but you can change the name by clicking on the sound bank or by right-clicking the sound bank and selecting Rename. The sound bank window is also added to the XACT workspace to the right.

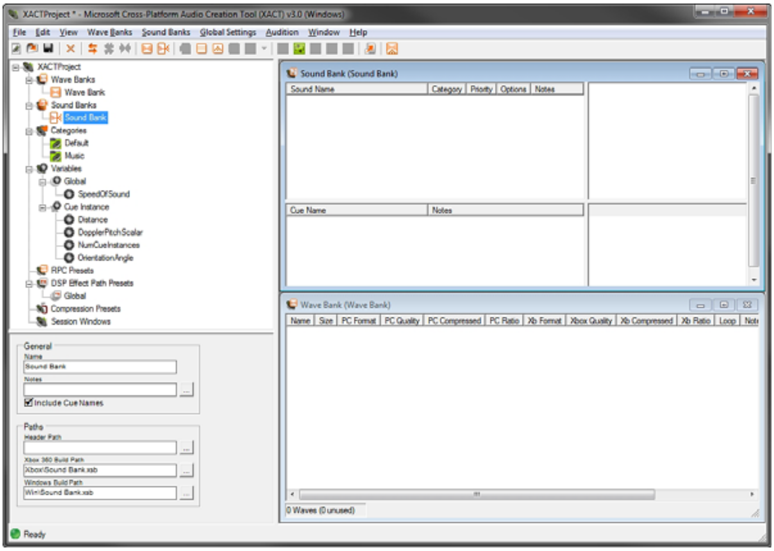
The workspace might be getting cluttered now, so the wave bank and sound bank windows might be overlapping. XACT provides a nice window-ordering option to organize your windows automatically. Click the Window menu option to display the windowing options. Select Tile Horizontally from the drop-down menu. The sound bank and wave bank windows should now organized as shown in Figure 13.3.

You can now add the source wave files to the project. Right-click the Wave Bank in the XACT project tree and select Insert Wave File(s). In the open file dialog box, choose a wave file and click the Open button.


XACTProject * - Microsoft Cross-Platform Audio Creation Tool (XACT) v3.0 (Windows)
File Edit View Wave Banks Sound Banks Global Settings Audition Window Help
XACTProject
Wave Banks
Wave Bank
Sound Banks
Categories
Default
Music
Variables
Global
SpeedOfSound
Cue Instance
Distance
DopplerPitchScalar
NumCueInstances
OrientationAngle
RPC Presets
DSP Effect Path Presets
Global
Compression Presets
Session Windows
General
Paths
Wave Bank (Wave Bank)
Name
Size
PC Format
PC Quality
PC Compressed
PC Ratio
Xb Format
Xbox Quality
Xb Compressed

Name: Wave Bank
Notes:
Type: In Memory | Streaming
Size: 0
☐ Friendly Names ☑ Sync In-Game Data
Compression Preset: <none>
Header Path:
Xbox 360 Build Path:
0 Waves (0 unused)
Ready

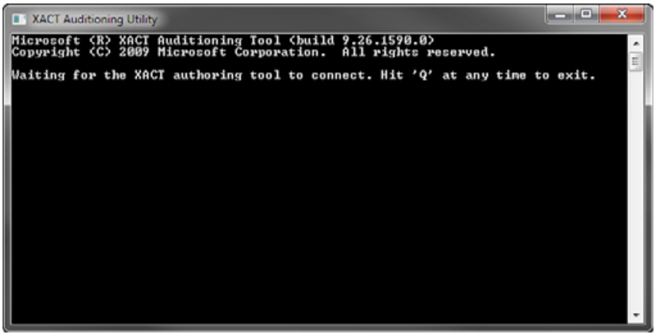
Figure 13.2 Wave Bank window in the XACT tool


XACTProject * - Microsoft Cross-Platform Audio Creation Tool (XACT) v3.0 (Windows)
File Edit View Wave Banks Sound Banks Global Settings Audition Window Help
XACTProject
Wave Banks
Wave Bank
Sound Banks
Sound Bank
Categories
Default
Music
Variables
Global
SpeedOfSound
Cue Instance
Distance
DopplerPitchScalar
NumCueInstances
OrientationAngle
RPC Presets
DSP Effect Path Presets
Global
Compression Presets
Session Windows
General
Paths
Sound Bank (Sound Bank)
Sound Name
Category
Priority
Options
Notes
Cue Name
Notes
Wave Bank (Wave Bank)
Name
Size
PC Format
PC Quality
PC Compressed
PC Ratio
Xb Format
Xbox Quality
Xb Compressed
Xb Ratio
Loop
Note

Name: Sound Bank
Notes:
☑ Include Cue Names
Header Path:
Xbox 360 Build Path: Xbox\Sound Bank.xsb
Windows Build Path: Win\Sound Bank.xsb
0 Waves (0 unused)
Ready

Figure 13.3 Sound and wave bank windows

To hear the newly added wave file play, launch the XACT Auditioning Utility. This tool can be found in the same Tools folder under Microsoft XNA Game Studio 4.0 as the XACT tool is located. Launching the XACT Auditioning Utility should bring up a console window like the one in Figure 13.4.



```
Microsoft (R) XACT Auditioning Tool (build 9.26.1590.0)
Copyright (C) 2009 Microsoft Corporation.  All rights reserved.

Waiting for the XACT authoring tool to connect. Hit 'Q' at any time to exit.
```

Figure 13.4 XACT Auditioning Utility

To listen to the wave file you just added to the wave bank, right-click the entry and select Play.

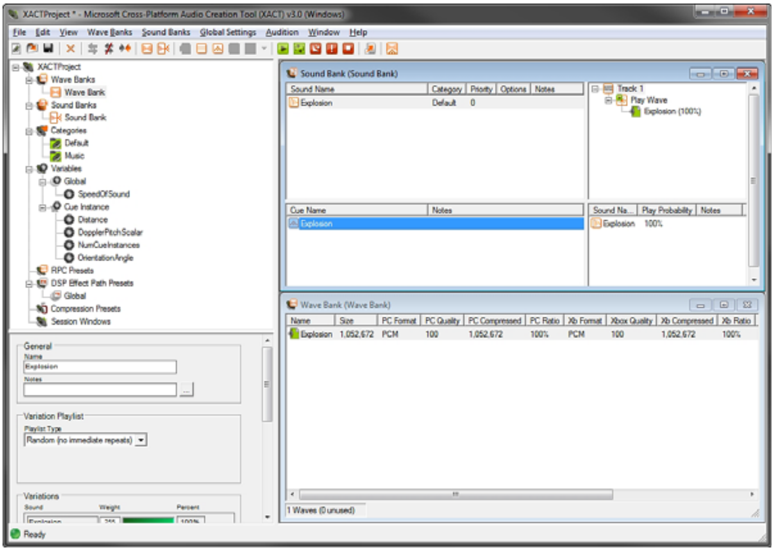
Note

If you receive and error message that says, "Could not connect to a Windows auditioning server," verify that you have started the XACT Auditioning Utility tool.

Finally, you need to set up a cue. The developer uses the cue to play back a sound or a group of sounds. The cue is stored and accessed from the sound bank.

The quickest way to add a cue is by dragging a wave from the wave bank onto the cue section of the sound bank windows, which is in the lower left corner (see Figure 13.5).

After you drag the wave file and create the cue, notice there are a number of new items that display in the sound bank window. Because a new sound has been created, by default, it is created with the same name as the wave. The sound is displayed in the top left of the sound bank window. After selecting the sound entry, there is a track listing to the right of the window. Each sound can contain a number of tracks. In this case, a single track is created that plays the wave from the wave bank. Below the sound bank is the cue window that you dragged the wave to. A new cue is created using the same name as the wave that was used to create it. As we discussed, the cue is used in the game code to control the playback of the sounds. So, a cue is required if you want to cause the sound to be played in your game. A cue is made up of multiple sounds from the sound bank entries in the window above it. Because the cue is created by using the dragging shortcut, the sound of the same name is added to the cue. By right-clicking the cue, you can select the Play menu option to hear the cue play.


XACTProject * - Microsoft Cross-Platform Audio Creation Tool (XACT) v3.0 (Windows)
File Edit View Wave Banks Sound Banks Global Settings Audition Window Help
XACTProject
Wave Banks
Wave Bank
Sound Banks
Sound Bank
Categories
Default
Music
Variables
Global
SpeedOfSound
Cue Instance
Distance
DopplerPitchScalar
NumCueInstances
OrientationAngle
RPC Presets
DSP Effect Path Presets
Global
Compression Presets
Session Windows
General
Variation Playlist
Variations
Sound Bank (Sound Bank)
Track 1
Play Wave
Explosion (100%)
Wave Bank (Wave Bank)
1 Waves (0 unused)
Ready

Name: Explosion

Playlist Type: Random (no immediate repeats)

Sound Name	Category	Priority	Options	Notes
Explosion	Default	0		

Cue Name	Notes
Explosion	

Sound Na...	Play Probability	Notes
Explosion	100%	

Name	Size	PC Format	PC Quality	PC Compressed	PC Ratio	Xb Format	Xbox Quality	Xb Compressed	Xb Ratio
Explosion	1,052,672	PCM	100	1,052,672	100%	PCM	100	1,052,672	100%

Figure 13.5 XACT windows after dragging wave to the cue window

> **Note**
>
> Without using the cue-dragging shortcut, multiple steps are required to set up a new cue for playback. First, you need to create a new sound, add a track to the sound, and then add the wave to the track. Next, create a new cue and add the sound to the cue.

Now you are ready to switch over to the code to show you how to play the cue you just created. Save your project and open XNA Game Studio.

Sound Playback Using Cue

It's time to play the cue you just created in the XACT tool. You need to add the XACT project you created to the content project of your game. Just like adding other types of content, it is easy to add the XACT project. Right-click the content project and select Add, and then click Existing Item. Then, select the .xap XACT project file in the dialog box and click the Add button. Building the content project now should show that three files are built from the single .xap file. XACTProject.xgs is the audio engine file and is named after the project name that was used for your XACT project. The Wave Bank.xwb file contains the data from the wave bank of the same name. If you renamed the wave bank in your XACT project, this file uses the name you used for your wave bank. The Sound Bank.xsb file contains the data from the sound bank in your XACT project. Like the wave bank, the name of the file is the same as you used for the sound bank in the XACT project and is set to whatever you named your sound bank.

In your game, add the following member variables for the `AudioEngine`, `SoundBank`, `WaveBank`, and `Cue`:

```
AudioEngine audioEngine;
SoundBank soundBank;
WaveBank waveBank;
Cue cue;
```

The `AudioEngine` is used to perform necessary updates to the underlying audio system. The `AudioEngine.Update` method is required to be called once per frame to ensure the audio playback stays up to date and is not stalled. The other three variables for the `SoundBank`, `WaveBank`, and `Cue` are the code counterparts to the ones created in the XACT project.

To load the files that were built by processing the XACT project through the content pipeline, add the following lines of code to your games `LoadContent` method:

```
// Load file built from XACT project
audioEngine = new AudioEngine("Content\\XACTProject.xgs");
waveBank = new WaveBank(audioEngine, "Content\\Wave Bank.xwb");
soundBank = new SoundBank(audioEngine, "Content\\Sound Bank.xsb");
// Get an instance of the cue from the XACT project
cue = soundBank.GetCue("Explosion");
```

Finally in the games `Update` method, check to see whether the spacebar is pressed. If the spacebar is pressed, the cue plays:

```
// Play the cue if the space bar is pressed
if (currentKeyboardState.IsKeyDown(Keys.Space) &&
    lastKeyboardState.IsKeyUp(Keys.Space))
{
    cue.Play();
}

// Update the audio engine
audioEngine.Update();
```

After playing the sound, call the `AudioEngine.Update` method to ensure it is called each frame.

A `Cue` has similar playback methods as `SoundEffectInstance`, such as `Play`, `Pause`, `Resume`, and `Stop`. There is also an `Apply3D` method, which takes an `AudioListener` and `AudioEmitter`.

After `Play` is called on the `Cue`, it can't be called again. You can call `Pause` and then `Resume`, but calling `Play` again will cause an `InvalidOperationException`. To play the sound more than once, create a new `Cue` instance by calling the `SoundBank.GetCue` method. Each `Cue` returned from `GetCue` is its own instance. This also means you should not call `GetCue` unless you need a new instance since it will allocate memory.

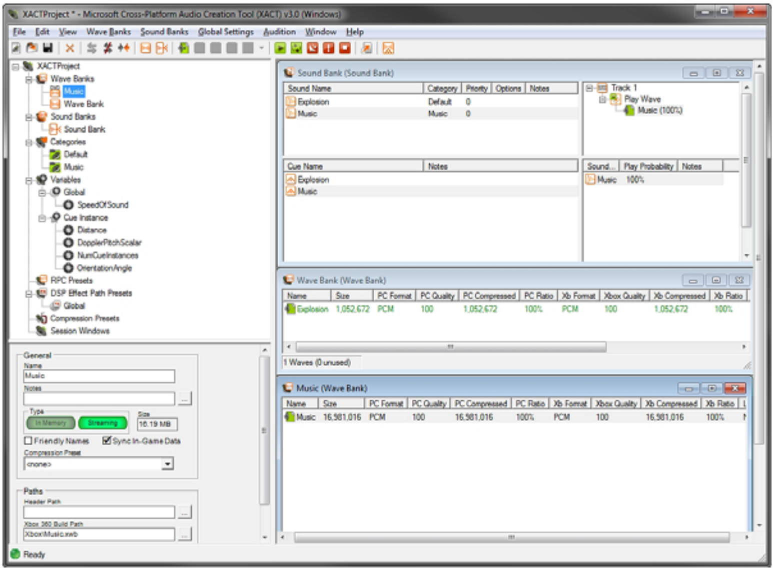
You can use the `SoundBank.PlayCue` method to just play a cue and have no other controls on the cue similar to how fire and forget sound effects work. You can use the following code to play a cue without creating an instance of a `Cue`:

```
// Play cue without cue instance
if (currentKeyboardState.IsKeyDown(Keys.P) &&
    lastKeyboardState.IsKeyUp(Keys.P))
{
    soundBank.PlayCue("Explosion");
}
```

Streaming

Some wave files are large, especially the wave files that are used to play back music in your game. By default, the wave files used in the XACT project are loaded into memory at runtime. This can cause problems if your file is quite large because the file might not fit into memory or it might take up more space than what you would like. Streaming is set at the wave bank level for all of the wave files contained within the wave bank.

Let's add some streaming music to the previous XACT sample. Open the XACT project you created previously. Add a new wave bank and call it Music. Select the new wave bank in the XACT project tree. The properties of the wave bank display in the lower left window. In the Type section, select the Streaming button. It should be highlighted a green color (see Figure 13.6).


XACTProject * - Microsoft Cross-Platform Audio Creation Tool (XACT) v3.0 (Windows)
File Edit View Wave Banks Sound Banks Global Settings Audition Window Help
XACTProject
Wave Banks
Music
Wave Bank
Sound Banks
Sound Bank
Categories
Default
Music
Variables
Global
SpeedOfSound
Cue Instance
Distance
DopplerPitchScalar
NumCueInstances
OrientationAngle
RPC Presets
DSP Effect Path Presets
Global
Compression Presets
Session Windows
Sound Bank (Sound Bank)
Track 1
Play Wave
Music (100%)
Wave Bank (Wave Bank)
Music (Wave Bank)
In Memory
Streaming
Ready

Sound Name	Category	Priority	Options	Notes
Explosion	Default	0		
Music	Music	0		

Cue Name	Notes
Explosion	
Music	

Sound...	Play Probability	Notes
Music	100%	

Name	Size	PC Format	PC Quality	PC Compressed	PC Ratio	Xb Format	Xbox Quality	Xb Compressed	Xb Ratio
Explosion	1,052,672	PCM	100	1,052,672	100%	PCM	100	1,052,672	100%

1 Waves (0 unused)

Name	Size	PC Format	PC Quality	PC Compressed	PC Ratio	Xb Format	Xbox Quality	Xb Compressed	Xb Ratio
Music	16,981,016	PCM	100	16,981,016	100%	PCM	100	16,981,016	100%

General
Name: Music
Notes:
Type: Streaming
Size: 16.19 MB
☐ Friendly Names ☑ Sync In-Game Data
Compression Preset: <none>

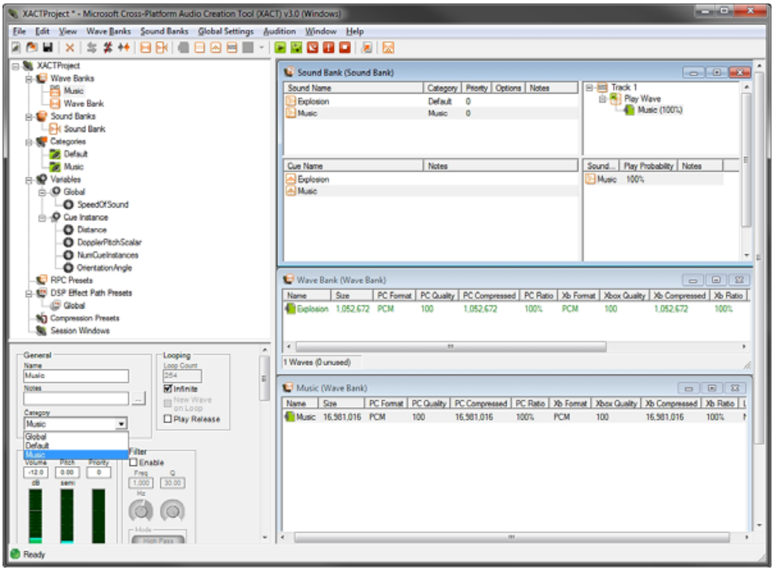
Paths
Header Path:
Xbox 360 Build Path: Xbox\Music.xwb

Figure 13.6 Setting wave bank to streaming

To add the music source files to the wave bank, right-click the wave bank and select Insert Wave File(s). Select your music file in the dialog window and click the Open button. Now that the source wave file is added to your streaming wave bank, you need a cue to use for playback. Like you did before, drag your music wave into the sound bank's cue window to create the cue for your music wave.

Finally, you change the sound that was created for the music cue to mark it as background music. Setting a sound category to Music allows users on the Xbox 360 to play their own music library over your game's music. This allows the user to hear the sound effects in your game but to override the music to what they are playing using the media playback provided on the Xbox 360.

To change the sound category, select the music sound entry that you just created in the sound bank. In the lower left, select the Category drop-down and select Music (see Figure 13.7). Next, select the Infinite checkbox under the Looping section to have the sound continually play until it is stopped. Save the XACT project, so you can add the music to your previous code sample.


XACTProject * - Microsoft Cross-Platform Audio Creation Tool (XACT) v3.0 (Windows)
File Edit View Wave Banks Sound Banks Global Settings Audition Window Help
XACTProject
Wave Banks
Music
Wave Bank
Sound Banks
Sound Bank
Categories
Default
Music
Variables
Global
SpeedOfSound
Cue Instance
Distance
DopplerPitchScalar
NumCueInstances
OrientationAngle
RPC Presets
DSP Effect Path Presets
Global
Compression Presets
Session Windows
Sound Bank (Sound Bank)
Track 1
Play Wave
Music (100%)
Wave Bank (Wave Bank)
1 Waves (0 unused)
Music (Wave Bank)
Volume
Pitch
Priority
Filter
Mode
High Pass
Ready

General
Name: Music
Notes:
Category: Music
Global
Default
Music

Looping
Loop Count: 254
☑ Infinite
☐ New Wave on Loop
☐ Play Release

Volume -12.0 dB
Pitch 0.00 semi
Priority 0

☐ Enable
Freq 1,000 Hz
Q 30.00

Sound Name	Category	Priority	Options	Notes
Explosion	Default	0		
Music	Music	0		

Cue Name	Notes
Explosion	
Music	

Sound...	Play Probability	Notes
Music	100%	

Name	Size	PC Format	PC Quality	PC Compressed	PC Ratio	Xb Format	Xbox Quality	Xb Compressed	Xb Ratio
Explosion	1,052,672	PCM	100	1,052,672	100%	PCM	100	1,052,672	100%

Name	Size	PC Format	PC Quality	PC Compressed	PC Ratio	Xb Format	Xbox Quality	Xb Compressed	Xb Ratio
Music	16,981,016	PCM	100	16,981,016	100%	PCM	100	16,981,016	100%

Figure 13.7 Setting sound category to Music

You need two additional member variables to add streaming music to the previous example. Add the following two member variables:

```
WaveBank streamingWaveBank;
Cue musicCue;
```

Like before, you need to load the wave bank. After the other audio loading in the games `LoadContent` method, add the following lines of code:

```
// Load streaming wave bank
streamingWaveBank = new WaveBank(audioEngine, "Content\\Music.xwb", 0, 4);
// The audio engine must be updated before the streaming cue is ready
audioEngine.Update();
// Get sue for streaming music
musicCue = soundBank.GetCue("Music");
// Start the background music
musicCue.Play();
```

The overload of the `WaveBank` constructor takes an additional two arguments. The first is the `offset` into the wave to start. In most cases, you want to start at the beginning, so a value of 0 should be set. The last parameter is the `packetsize`, which determines the size of the streaming buffer used internally to store the file as it is loaded. It is in units of DVD sectors, which are 2,048 bytes each. The minimum value must be 2. The higher the number, the larger the memory buffer, which allows for smoother playback. You can test different values to determine the minimum value that smoothly streams the music in your game.

After creating a new streaming wave bank, you must call the `AudioEngine.Update` method to play a cue that contains the streamed data from the wave bank.

The last two steps are to get the `Cue` from the `SoundBank` and to call the `Play` method to start the background music. Now your game has streaming background music that can be overridden by the player.

Dynamic Sound Effects

Not all sounds have to come from a prerecorded file. New to XNA Game Studio 4.0 are the capabilities to record microphone data using the new `Microphone` class and playback of dynamic sound effect data using the new `DynamicSoundEffectInstance` class.

Recording Audio with a Microphone

The new Microphone type is supported on all of the XNA Game Studio 4.0 platforms. On Windows, this is any of the recording devices recognized by Windows including the Xbox 360 headset. Make sure the microphone is set up in Windows before trying to record using the XNA APIs. Xbox 360 supports the headsets along with the Xbox 360 wireless microphone. Windows Phone 7 supports reading the built in microphone as well as Bluetooth headsets.

Enumerating Microphones

To enumerate all of the recording devices available to your game, you can use the `Microphone.All` property. This returns a `ReadOnlyCollection` of `Microphone` objects.

Use the `Microphone.Name` field to determine the friendly name of the recording device. For example, the Xbox 360 wired headset connected to the Xbox 360 wired controller on a Windows 7 machine returns Headset Microphone (Headset (XBOX 360 For Windows)).

You can use the `Microphone.IsHeadset` property to determine whether a recording device is a wired headset or a Bluetooth hands-free headset.

Most of the time, you will not want to enumerate all of the recording devices and will just want to use the default `Microphone` that the user or platform has selected. You can use the `Microphone.Default` property to easily select the preferred default recording device for the particular platform.

```
Microphone microphone = Microphone.Default;
```

The `Microphone` has a specific state, which is returned by the `Microphone.State` property. The state is returned as a `MicrophoneState` enumeration, which has the values of `Started` and `Stopped`. The `Microphone` starts its `State` in the `Stopped` state. To start the recording process, you must call the `Microphone.Start` method. After the `Microphone` starts recording, it gathers buffered data that can then be used for playback. Use the `Microphone.Stop` method to stop the gathering of recording data.

Reading Microphone Data

After the `Microphone` starts, the internal buffer starts to fill up with the recorded values received from the recording device. The internal buffer size of the `Microphone` is exposed by the `Microphone.BufferDuration` property. This property can be used to read and set the buffer duration to use for recording. The value is specified as a `TimeSpan` and must be between 100 and 1000 milliseconds.

> **Note**
>
> The `Microphone.GetSampleDuration` method can be used to determine the `TimeSpan` a given sample buffer size supports.

The game needs a `byte` array buffer to store the data returned from the `Microphone`. To determine the size of the buffer, the `Microphone.GetSampleSizeInBytes` method calculates the required buffer size given a `TimeSpan`. The buffer should be large enough to hold all of the data the `Microphone` returns. If you use the `Microphone.BufferDuration` property as the parameter passed to `GetSampleSizeInBytes`, the size returned is equal to the internal buffer size:

```
int bufferSize = microphone.GetSampleSizeInBytes(microphone.BufferDuration);
byte[] buffer = new byte[bufferSize];
```

Now you have a buffer to read the `Microphone` data. While the `Microphone` is recording, an internal buffer continually updates with the latest data from the physical recording device. Access this internal buffer by using the `Microphone.GetData` method. This method returns the current buffer since the last `GetData` call. You can access the buffer in two different ways. You can read the current buffer each frame, which returns a varying amount of data depending on how long the last frame took:

```
int returnedData = microphone.GetData(buffer);
```

In this case, the `GetData` method returns the size of data updated in the buffer.

> **Note**
>
> A `GetData` overload enables you to specify an offset and size to use when filling the provided buffer.

The other way to use the `GetData` method is to only request the data after the internal buffer is full. The `Microphone.BufferReady` event is raised after the internal buffer is full and a full read can occur. In the `BufferReady` event handler, you can call `GetData` to access the full internal buffer, which is the same size as your created buffer if you used the `BufferDuration` as the parameter to the `GetSampleSizeInBytes` method:

```
microphone.BufferReady += new EventHandler<EventArgs>(OnBufferReady);
public void OnBufferReady(object sender, EventArgs args)
{
    // Read entire buffer
    microphone.GetData(buffer);

    // Process data
}
```

Playback Using DynamicSoundEffectInstance

Now that you know how to select a microphone, start recording, and read the recorded buffer data, let's play back the audio as it is recorded from the `Microphone`.

To play the recorded buffer data, use the new `DynamicSoundEffectInstance` class. `DynamicSoundEffectInstance` is derived from `SoundEffectInstance`. Like `SoundEffectInstance`, it allows for basic playback controls such as `Play`, `Pause`, `Stop`, and `Resume`. It also has properties for the `Pitch`, `Pan`, and `Volume`.

Add the following member variables to your game class to store the `Microphone`, buffer to store the recorded data, and `DynamicSoundEffectInstance` to play the recorded data:

```
Microphone microphone;
byte[] buffer;
DynamicSoundEffectInstance dynamicSoundEffectInstance;
```

In the game's `Initialize` method, add the following lines of code:

```
// Request the default recording device
microphone = Microphone.Default;
// Calculate the size of the recording buffer
int bufferSize = microphone.GetSampleSizeInBytes(microphone.BufferDuration);
buffer = new byte[bufferSize];
// Subscribe to ready event
```

```
microphone.BufferReady += new EventHandler<EventArgs>(OnBufferReady);
// Start microphone recording
microphone.Start();

// Create new dynamic sound effect to playback microphone data
dynamicSoundEffectInstance =
    new DynamicSoundEffectInstance(microphone.SampleRate,
                                    AudioChannels.Mono);
// Start playing
dynamicSoundEffectInstance.Play();
```

After `DynamicSoundEffectInstance` constructor takes two parameters, the first `sampleRate` takes the sample rate in Hertz of the audio content that passes to the `DynamicSoundEffectInstance`. The second parameter channel takes an `AudioChannels` enumeration value of `Mono` or `Stereo`. Because the microphone records in mono, use the `Mono` enumeration.

After you create the `DynamicSoundEffectInstance`, play the sound. The sound continues to play as you keep passing new buffer data to the instance.

Finally, you need to implement the `OnBufferReady` method that you passed to the `Microphone.BufferReady` event:

```
public void OnBufferReady(object sender, EventArgs args)
{
    // Read entire buffer
    microphone.GetData(buffer);
    // Send latest buffer to the dynamic sound effect
    dynamicSoundEffectInstance.SubmitBuffer(buffer);
}
```

This event will be called whenever the internal `Microphone` buffer is full and is ready to be read. The buffer is then read using the `Microphone.GetData` method, which fills the buffer you created. The new buffer then needs to be passed to the `DynamicSoundEffect.SubmitBuffer` method so playback continues.

> **Note**
>
> If you change the `Pitch` of the `DynamicSoundEffectInstance` while using the microphone as the source, the recording rate differs from the playback rate. Playback falls further and further behind if you lower the `Pitch`, or it stops and waits for more data, causing a pop sound if you raise the `Pitch`. Either way, it makes your voice sound cool—so try it out!

Generating Dynamic Sound Effects

Along with using the `Microphone` to generate dynamic data to playback, you can also create the buffer dynamically. This means your game can create new and interactive sounds that are driven by user input or interesting algorithms. Because creating dynamic audio data could be a book by itself, we cover just the basics of how to generate some simple data. Feel free to experiment and research how to create new and interesting dynamic sound effects.

What we perceive as sound is actually changes in air pressure that cause bones in our inner ear to vibrate. This vibration is what we call sound. To generate sound, we need to create this change in air pressure. Most electronics use speakers to generate the change. Speakers convert a digital signal into sound by vibrating a diaphragm, which in turn vibrates the air around us.

The data that you generate also needs to cause a vibration. There are many different ways to generate vibration waves, but one of the easiest is using the sine wave.

Now let's generate some tones. Add the following member variables to your game:

```
const int SampleRate = 48000;
DynamicSoundEffectInstance dynamicSoundEffectInstance;
byte[] buffer;
int bufferSize;

// Frequency to generate
double frequency = 200;
// Counter to mark where we are in a wave
int totalTime = 0;
```

The `SampleRate` is the amount of samples that play in a second. The frequency is the amount of times the speaker vibrates per second. The faster the frequency, the higher pitch the tone sounds. Use the `totalTime` value to store where in the sine wave you are currently. The sine wave oscillates between –1 and 1 over the source of a period of two Pi.

Next in your games `Initialize` method, add the following lines of code:

```
// Create new dynamic sound effect and start playback
dynamicSoundEffectInstance = new DynamicSoundEffectInstance(SampleRate,
AudioChannels.Mono);
dynamicSoundEffectInstance.BufferNeeded += new
EventHandler<EventArgs>(OnBufferNeeded);
dynamicSoundEffectInstance.Play();

// Calculate the buffer size to hold 1 second
bufferSize =
dynamicSoundEffectInstance.GetSampleSizeInBytes(TimeSpan.FromSeconds(1));
buffer = new byte[bufferSize];
```

The previous code creates a new `DynamicSoundEffectInstance` with the previously defined `SampleRate` and only a single `Mono` audio channel. Use the `BufferNeeded` event to signal when the `DynamicSoundEffectInstance` is in need of more data to playback. Finally, start the instance by using the `Play` method.

Note

Stereo requires twice the amount of data to feed the left and right channels.

Calculate the `bufferSize` by using the `GetSampleSizeInBytes` method to determine how much data is needed for a single second.

The final section of code is what generates the tone buffer to play. Add the following method to your game:

```
// Generate sound when needed
void OnBufferNeeded(object sender, EventArgs e)
{
    // Loop over entire buffer
    for (int i = 0; i < bufferSize - 1; i += 2)
    {
        // Calculate where we are in the wave
        double time = (double)totalTime / (double)SampleRate;
        // Generate the tone using a sine wave
        short currentSample = (short)(Math.Sin(2 * Math.PI * frequency * time) *
                                    (double)short.MaxValue);

        // Store the generated short value in byte array
        buffer[i] = (byte)(currentSample & 0xFF);
        buffer[i + 1] = (byte)(currentSample >> 8);

        // Incrament the current time
        totalTime += 2;
    }

    // Submit the buffer for playback
    dynamicSoundEffectInstance.SubmitBuffer(buffer);
}
```

To generate the tone, loop over the entire buffer. Although the buffer is in bytes, the data for each channel of the `DynamicSoundEffectInstance` is 16 bits or a short. Because you use `Mono`, each loop covers two of the bytes in the buffer.

In each iteration of the loop, the short value for the sine wave is calculated. This calculation takes into account both the current time and the frequency. The time has to take into account how many samples occur per second because you want the frequency value to also be in hertz. The short value calculated from the `Math.Sin` method is in the range of –1 to 1. Because you want the value between the minimum and maximum values for a short, multiply the value by `short.MaxValue`.

The generated short value then needs to be broken into the high and low order bytes so it can be added to the buffer array. The time value is then incremented by two because you move through the array by two.

The final step is to call `SubmitBuffer` on the `DynamicSoundEffectInstance` to supply it with the latest buffer. If you run the sample code, you hear a nice solid tone.

Although the sine wave is a simple example of how to generate dynamic sound effects, it is the building block for many more complex effects. Spend some time trying to generate other types of sound waves and even mix them together.

Summary

Your game should be much louder now. We covered the different ways to play audio in XNA Game Studio 4.0. We covered loading individual sound effect files and using the `SoundEffect` APIs to play them back. We covered using the properties of the `SoundEffectInstance` to change how the sounds are played back, changing the `Pitch`, `Pan`, `Volume`, and changing their position in 3D space.

We also covered how to use the XACT tool to add wave banks, sound banks, and cues to your game. We demonstrated how to stream and mark audio as music so you can play large music files that the game player can override with his or her own music selection.

Finally, we covered dynamically generated audio. We showed how to use the `Microphone` to read an audio buffer and how to use the `DynamicSoundEffectInstance` to play back the recorded buffer. Then we generated audio using a sine wave to play a solid steady tone.

14

Storage

In the majority of games that are made these days, it is rare that the entirety of the game can be completed in a single sitting of a reasonable length. Some games can last beyond 40 hours, and although sitting around playing games nonstop for that amount of time sounds fun, everyone needs a break sometime! The majority of games now have a concept of saving state for you to continue where you left off. In this chapter you will learn about the various ways to use storage to do this including:

- Using isolated storage
- Using the XNA Game Studio storage API
- Loading loose files in your games

What Is Storage?

Although many games incorporate saving state directly into game play (for example, by having only certain places you can save progress), this chapter focuses solely on how to use the facilities provided by the XNA runtime and the platforms on which it runs.

XNA Game Studio 4.0 comes with two separate storage solutions: one for the Reach profile that runs on the platforms XNA runs on and another for the HiDef profile that runs only on Xbox and Windows and gives you extra control for your Xbox storage needs.

Isolated Storage

The first type of storage you learn about here has actually been around for a while, and it is called Isolated Storage. As the name implies, things stored here are isolated from other things—in this case, other applications. You are safe in the knowledge that anything you write in your store, no other applications can read or write. Let's start off by creating a new Windows Phone Game project to show some of the features that this storage API has.

Before you can store anything, though, you need something interesting to save. You can imagine that this could be some state of a level such as where the player is, the amount of health they have, and fun stuff like that; however, for this example, you need something simple to show the concepts.

First, add a `SpriteFont` object called `Font` to your project because you use this to render the data you will save. Next, you need to add a new using statement to the top of your `game1.cs` code file because the default template that is used for creating new projects doesn't actually have the namespace that the isolated storage constructs live in. Add this to the top of your code file:

```
using System.IO;
using System.IO.IsolatedStorage;
```

What this example does is to allow a user to plot points for the game, and then save those points for later. You need a few variables for this, so go ahead and add these to your game now:

```
SpriteFont font;
List<Vector2> points;
```

As you see, the points are just their positions stored in a vector. You need an easy way for the user to add new points, though, so in your `Initialize` method, add the following to tell the system that you will listen for a `Tap` gesture from the touch panel (which is simulated with the mouse on the emulator):

```
// Support tap gestures to map points
TouchPanel.EnabledGestures = GestureType.Tap;
points = new List<Vector2>();
```

Also notice that you initialize the points list. Because you need to render these points somehow, you also need to create your `SpriteFont` object in the `LoadContent` method:

```
font = Content.Load<SpriteFont>("font");
```

Of course, you need to draw these points. You can use a simple x character to show the position of the points, so add the following code to your `Draw` overload before the call to `base.Draw`:

```
spriteBatch.Begin();
foreach (Vector2 position in points)
{
    spriteBatch.DrawString(font, "x", position, Color.White);
}
spriteBatch.End();
```

You don't need anything super fancy here; you simply draw a white x at every position in the points list. All that is left now is to add points to the list, luckily that's quite easy! Just add this code to your `Update` overload:

```
while (TouchPanel.IsGestureAvailable)
{
    GestureSample gesture = TouchPanel.ReadGesture();
    points.Add(gesture.Position);
}
```

This is extremely simple stuff here. Simply read all gestures while any are available and add the position of that tap to your list. Running the game now lets you tap on the screen to get a bunch of white X's drawn everywhere, much like you'd see in Figure 14.1.

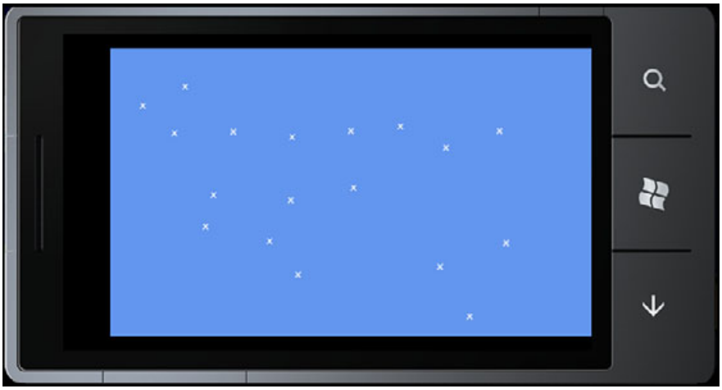


Figure 14.1 A bunch of white X on the emulator

As you'd probably guess, if you stop and restart the application, all of your wonderful x marks get lost. What if you designed a great work of art using small x marks? This situation needs to be rectified.

Saving and Loading Data

First, you need a good spot in your game to save the data. In a real game, this would probably be at the end of a level or at a defined save point in the game. Because this is just an example, though, you can instead pick an arbitrary point, in this case, when the game exits. Add the following override to your game:

```
protected override void OnExiting(object sender, EventArgs args)
{
    // Save our data
    using (IsolatedStorageFile file =
➥IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (BinaryWriter writer = new
➥BinaryWriter(file.CreateFile("pointData")))
        {
            foreach (Vector2 v in points)
            {
                writer.Write(v.X);
                writer.Write(v.Y);
            }
        }
    }
    base.OnExiting(sender, args);
}
```

The first thing to notice is the creation of an `IsolatedStorageFile` object via the `GetUserStoreForApplication` method. You can think of this object as the root of any storage facilities you need. It can contain files, directories, and all kinds of things you see in a few moments. Next, you want to use the `CreateFile` method to create a new file in your isolated store. The name of this file can be any valid filename you would like; for this example, choose `pointData`.

Because the `CreateFile` method returns a stream, you can use the `BinaryWriter` helper object to allow you to easily write data into your store. You simply enumerate over each of the points in your list and write out the `X` and `Y` components to the stream. These statements are wrapped in the `using` keyword, so the objects are disposed (and closed) immediately when you're done with them.

Running the game now lets you continue to add new x marks and save the current state of the marks when you exit; however, you never read them back in, so you need to fix that. Add this override to your code (this happens when your game first starts up).

```
protected override void OnActivated(object sender, EventArgs args)
{
    // Load the data
    using (IsolatedStorageFile file =
➥IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (BinaryReader reader = new BinaryReader(file.OpenFile("pointData",
            FileMode.OpenOrCreate)))
        {
            for(int i = 0; i < reader.BaseStream.Length / 8; i++)
            {
                points.Add(new Vector2(reader.ReadSingle(),
➥reader.ReadSingle()));
            }
        }
    }
    base.OnActivated(sender, args);
}
```

In this call, you essentially reverse what you did when you saved the data. You once again create your `IsolatedStorageFile` object, but instead of creating a new file, you open an existing file. Naturally, you use the same name you used earlier when you saved the data. If you notice, in this example, you use the `FileMode.OpenOrCreate` option when opening the file. This opens the file if it exists or creates a new one if it does not, and you do this so you don't have to worry about an exception trying to open a file that doesn't exist. If you use the `FileMode.Open` option instead, the method throws an exception if that file does not exist.

Instead of using the `BinaryWriter` helper class you used during the save operation, you can instead use the `BinaryReader` helper class to get that data out. Because you know that you are writing pairs of floats from your vectors, you can calculate how many points are in this file by taking the total length (in bytes) and dividing by eight (two floats that are four bytes each). You then add each new point into your point list.

With that, when you start up the application, all of the points you had drawn on the screen before you exited appear back where you left them!

The IsolatedStorageFile Object

At a basic level, this small example actually covers everything you need to do in a game to save state. However, it doesn't give a great in-depth discussion of the features of the `IsolatedStorageFile` object.

You've seen the `OpenFile` method and the `CreateFile` method, although the `CreateFile` method is simply a helper method that calls `OpenFile` with the `FileMode.CreateNew` option passed in. These methods each return a `Stream` object that you can use however you want. Well, that's not entirely true. One of the overloads of `OpenFile` enables you to pass in a `FileAccess` enumeration that enables you to specify whether you want the file to be opened for `Read`, `Write`, or `ReadWrite`. If you've passed in `FileAccess.Read` to your `OpenFile` method, you cannot write to the stream that is returned, for example.

There are other things you can get from this object. If you need to know the amount of available space remaining for your files, you can use the `AvailableFreeSpace` property. If you want to completely delete everything in the store for your game, use the `Remove` method. That's a dangerous operation, though, so make sure you want to delete everything!

If you want to delete a single file rather than your entire store, use the `DeleteFile` method. For example, if you want to delete the file you created during the previous example, call `file.DeleteFile("pointsData")`.

You might notice that this store looks much like a file system. There are methods such as `CreateDirectory` and `CreateFile`. Having the isolated store mirror a file system enables you to store your data in a hierarchy if you have the need to do so. For example, if you store a high-score list for every level in your game, you might envision a list of directories for each level with each directory containing a file called `highscore.txt` that contains the high scores for that level. There's nothing stopping you from creating a bunch of files called `level1highscore.txt`, `level2highscore.txt`, and so on, but the option to have directories is there if you want to use it. With the ability to create a directory, the ability to remove it via the `DeleteDirectory` also exists.

If you imagine a scenario where you save the data for a particular character in your game and that character can be named by the player, you might be tempted to store the data in a file of that user-defined name. Because you don't know what that name is, you need a way to enumerate files (and directories) within your store, and luckily you can do that with the `GetDirectoryNames` and `GetFileNames` methods.

With the basics of the `IsolatedStorageFile` object understood, there is one more scenario in which you can use this concept, and that is tombstoning, which is discussed in depth in Appendix D, "Dealing with Tombstoning."

Although the isolated storage functionality exists on all the platforms that XNA runs on (and is part of the Reach profile), it lacks some of the features users have come to expect when running on some platforms, such as Xbox 360. Next, you'll learn how to access those features!

XNA Game Studio Storage

If you used previous versions of XNA Game Studio, this next section should be familiar. It is the updated version of the storage API that shipped for the last few releases. Although it is more complicated than the version you just saw, it also has more features. First, emulate what you did originally in your Windows Phone 7 project by allowing the user to draw x marks on the screen, save, and reload them. Create a new Xbox 360 Game project.

Recreating the Project on Xbox

Because you need to use the storage api, as well as various IO functions, update the using statements at the beginning of your new game1.cs code file:

```
using System.IO;
using Microsoft.Xna.Framework.Storage;
```

You need the sprite font again to render the data, so add a new sprite font to your content project named Font. Include the variables for your list of points and the following font:

```
List<Vector2> points;
SpriteFont font;
```

Add the initialization of the list in the `Initialize` overload:

```
points = new List<Vector2>();
```

Also, include the creation of the sprite font in your `LoadContent` overload:

```
font = Content.Load<SpriteFont>("font");
```

To render your points, add the following code before the `base.Draw` call in your `Draw` overload:

```
spriteBatch.Begin();
foreach (Vector2 position in points)
{
    spriteBatch.DrawString(font, "x", position, Color.White);
}
spriteBatch.End();
```

The code that you used to add new points to the screen in the Windows Phone project unfortunately does not work here. There is no touch screen for your Xbox 360! So instead, let users input points with the controller by allowing them to move a cursor around the screen and then pressing the A button to add points. To implement this simple feature, add a couple new variables to your project:

```
Vector2 currentPos;
GamePadState lastState;
```

The first variable is the current position of the onscreen cursor, and the second is the last frame's state of the controller. Initialize the current position variable in your initialization method as well:

```
currentPos = Vector2.Zero;
```

Note

Yes, it is true that you don't need to do this initialization (because the variable is by default initialized to zero), but it's a good habit to initialize everything in your initialization methods to ensure that objects have the correct values.

To draw the cursor on the screen so users can see where they will place points, add the following line to your project before the `End` call on `spriteBatch`:

```
spriteBatch.DrawString(font, "*", currentPos, Color.Red);
```

To allow the user to update its position, add the following code to the `Update` method:

```
GamePadState state = GamePad.GetState(PlayerIndex.One);
if (state.IsButtonUp(Buttons.A) && (lastState.IsButtonDown(Buttons.A)))
{
    // Add a new point here
    points.Add(currentPos);
}
// Move the current cursor
currentPos.X += (state.ThumbSticks.Left.X * 10);
currentPos.Y -= (state.ThumbSticks.Left.Y * 10);
// Don't let it go beyond the screen
if (currentPos.X < 0)
    currentPos.X = 0;
if (currentPos.Y < 0)
    currentPos.Y = 0;
if (currentPos.X > GraphicsDevice.Viewport.Width)
    currentPos.X = GraphicsDevice.Viewport.Width;
if (currentPos.Y > GraphicsDevice.Viewport.Height)
    currentPos.Y = GraphicsDevice.Viewport.Height;
lastState = state;
```

Note

This code assumes you have a controller plugged in and registered as player one. A real game handles this more appropriately.

The code here is pretty self-explanatory; it adds a new point to the list (at the current position) if the A button was pressed in the last frame and released in this frame. It increments the X member of the current position by how much you press the left thumbstick (multiplying by 10 makes the cursor move faster). It does the same thing with the Y member, except it decrements instead because the Y axis on the stick goes from 1 to –1. It then does some basic bounds to ensure that the cursor stays within the viewport, and stores the last state of the gamepad.

With that portion of the code done, save the data when you exit and restore it when you launch.

Devices and Containers

Let's take a few moments to discuss the `StorageDevice`, because it is one of the sources of the most common mistakes when dealing with storage on the Xbox. The Xbox 360 provides a number of places to store data. You might have a hard drive, you might have one or more memory units, and now you can even have USB memory sticks attached. Each one of these is considered a device and can be used to store data for your games. Notice an example of a list of devices in which to choose in Figure 14.2.

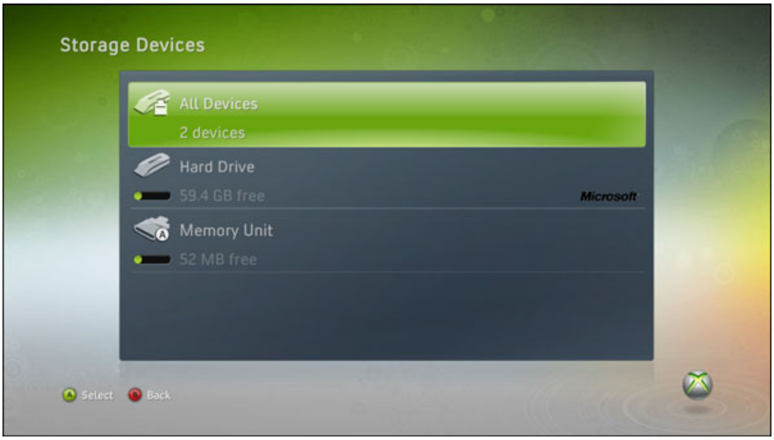


Figure 14.2 Devices on an Xbox 360

Each device can have a wide variety of containers in it, and much like on the phone, each game has all of its data segregated from all other games. When navigating to the system memory blade in the Xbox 360 dashboard, you can see how much data each game is storing, such as in Figure 14.3.

`StorageContainer` objects are the equivalent of the `IsolatedStorageFile` object you used previously. The big difference is that a game can have multiple containers, where each game is limited to a single isolated storage store. As you can see in Figure 14.4, this single game has three distinct storage containers in it.

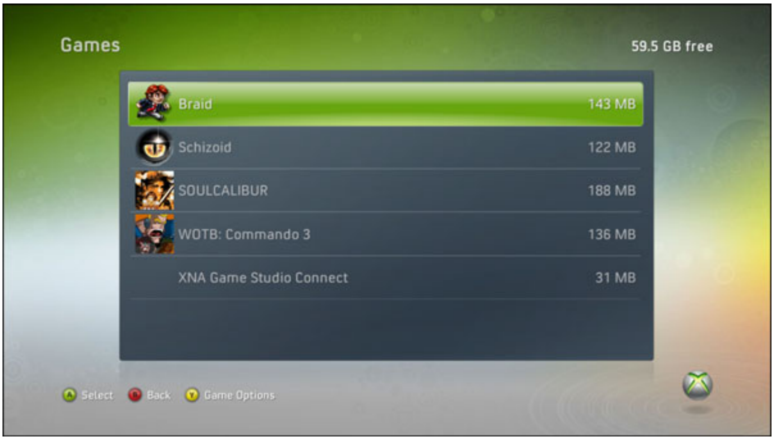

Games
59.5 GB free
Braid 143 MB
Schizoid 122 MB
SOULCALIBUR 188 MB
WOTB: Commando 3 136 MB
XNA Game Studio Connect 31 MB
Select
Back
Game Options

Figure 14.3 Game data on an Xbox 360

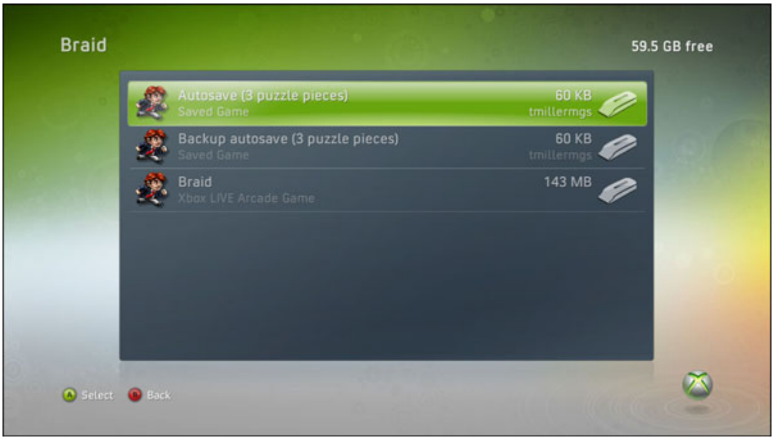

Braid
59.5 GB free
Autosave (3 puzzle pieces) 60 KB
Saved Game tmillermgs
Backup autosave (3 puzzle pieces) 60 KB
Saved Game tmillermgs
Braid 143 MB
Xbox LIVE Arcade Game
Select
Back

Figure 14.4 Multiple storage containers per game

When you view this screen on the dashboard, it shows the containers that were created for the game, not the files themselves. Each container can contain many files and folders just like the isolated storage files could earlier this chapter.

Getting a Device

Getting the device to save your data to is quite error prone. What makes this situation so easy to get wrong is how the user chooses where to save the data. It is expected that all Xbox games enable the user to choose where to save data, so the API has to enable him or her to do so. The problem arises because the system requires the game to continue to run (and to continue to draw) in order to display the Guide and enable the user to choose the device, but the API to pick the device blocks whatever thread on which it is running. If the thread happens to be the same one drawing, your game hangs. What's even worse though, if you have only a single device on the system, the Guide does not show at all. This means you can write code that hangs your game without even knowing it, which is what many people unfortunately do.

The API to get the device follows the common .NET async pattern, which is a hint to developers that this API needs to be performed asynchronously. With that small preamble out of the way, let's implement data storage for this example. First, declare a variable for your device:

```
StorageDevice storageDevice;
```

Now, you might wonder why you are going to store the device for this example when you didn't at first with the phone example. This is because obtaining the device can possibly force a UI popup to appear, and you don't want to ask the user multiple times, "Hey, where do you want to store this data?" So long as the device remains valid, you should continue to use it.

Now, create the device. Because this should be done at startup, do the loading of the data if it exists then as well. Add the following code to your game's `Initialize` method:

```
// Load the data
StorageDevice.BeginShowSelector(new AsyncCallback(
    (result) =>
    {
        storageDevice = StorageDevice.EndShowSelector(result);
        if (storageDevice != null)
        {
            storageDevice.BeginOpenContainer("GameData", new AsyncCallback(
                (openResult) =>
                {
                    using (StorageContainer file =
                        storageDevice.EndOpenContainer(openResult))
                    {
                        using (BinaryReader reader = new
                        BinaryReader(file.OpenFile("pointData")))
                        {
                            for (int i = 0; i < reader.BaseStream.Length / 8; i++)
                            {
                                points.Add(new Vector2(reader.ReadSingle(),
                                    reader.ReadSingle()));
                            }
                        }
                    }
                }
            ), null);
        }
    }
), null);
```

This was certainly much more in depth (and complicated looking) than the isolated storage version! Looks are a little deceiving though, because the majority of the complication is handling the async pattern that is used to both get the device and the container. Note that when you use the `AsyncCallback` (such as this example), the callback happens on a separate thread from the one on which you called the method. This enables the code to work if the Guide pops up because the main thread is not blocked. After you have a container, the code is identical to the isolated storage code you wrote previously.

Again, you might ask yourself why you're doing this in the constructor for this example, when during the phone example you did it in the `OnActivating` override. Unlike the phone, which doesn't have the concept of the game not the focus (since it is killed), the Xbox 360 does. When your game is not focused, it gets the `Deactivated` event, and when it regains focus it gets the `Activated` event. Showing the Guide deactivates the game while it is up, and then reactivates it when it goes away. So essentially, if you use that override, you get stuck in an endless loop of creating the device.

Now, if you run the game, it doesn't actually run—you get an exception. The exception text is quite descriptive, telling you that you need a `GamerServicesComponent`. This is discussed in depth in the next chapter, so for now, do what is needed to get the example running. Now add the following to your components collection in your game's `Initialize` method:

```
Components.Add(new GamerServicesComponent(this));
```

Now when you run the example, it shows the Guide if you have more than one device available like you see in Figure 14.5. If you have a only single device available, it is automatically chosen for you.

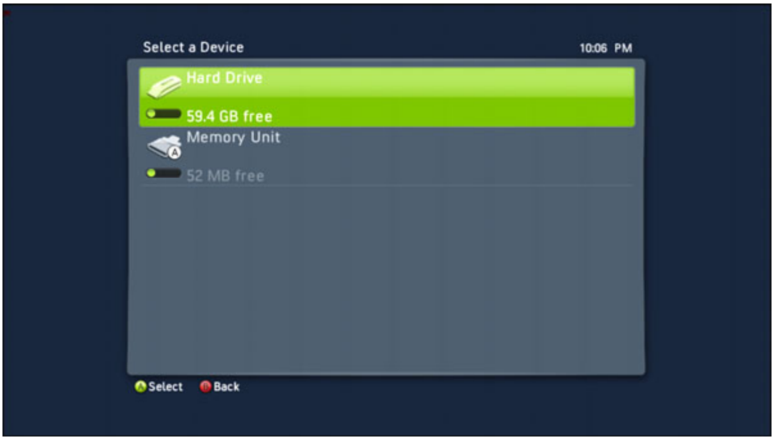


Figure 14.5 Choosing your storage device

> **Note**
>
> Notice also how you check whether the device returned is null. This is because the user can easily press the B button on the dialog and cancel choosing a device. In a real game, you want to send a warning about this and confirm that a user wants to continue without saving. Here, continue without saving.

Also, notice that the first parameter is a string in `BeginOpenContainer` calls. This is the friendly name of the container that users see when they look at the containers in the dashboard (as seen in Figure 14.4). This is, of course, the name of the container you open, so use the same string to get the same container again.

A common use for the containers is to separate data logically. For example, you might have each saved game stored in its own container. This gives the user the capability to manage the save game from the system user interface without being in the game.

Before getting into the details of the API, let's finish the example. Save the data before you leave the game, but this time use the following code for the `OnExiting` override:

```
protected override void OnExiting(object sender, EventArgs args)
{
    // Save our data
    if (storageDevice != null)
    {
        storageDevice.BeginOpenContainer("GameData", new AsyncCallback(
            (openResult) =>
            {
                using (StorageContainer file =
                    storageDevice.EndOpenContainer(openResult))
                {
                    using (BinaryWriter writer =
                        new BinaryWriter(file.CreateFile("pointData")))
                    {
                        foreach (Vector2 v in points)
                        {
                            writer.Write(v.X);
                            writer.Write(v.Y);
                        }
                    }
                }
            }
        ), null);
    }
    base.OnExiting(sender, args);
}
```

This is similar to before, in that you open the same container if your storage device isn't null, and then use the same code to write out the points you used earlier. Running the game now enables you to place points on the screen, exit, and run the game again to see them show back up.

Looking at the API

Now that you have the basic functionality down, let's look at the API a bit more in depth now. The `StorageDevice` has the two static methods you already used as well as a single event called `DeviceChanged`. Proper handling of this event can be crucial for your game.

Imagine that your player starts the game and chooses to save on the memory unit. Halfway through the game, the user pulls the memory unit out for some reason. Now you can no longer save, and worse yet, your game might crash! The `DeviceChanged` event is fired when any device is changed. To update this example to handle this case, add the following code to the end of your `Initialize` method:

```
StorageDevice.DeviceChanged += new EventHandler<EventArgs>(
    (o, e) =>
    {
        // The device changed, make sure my current device is still valid
        if (storageDevice != null)
        {
            if (!storageDevice.IsConnected)
            {
                // My device was disconnected, set it to null
                storageDevice = null;
                // Warn the user that we won't be saving now
                Guide.BeginShowMessageBox("No Longer Saving",
                    "The device you were using to save no longer exists " +
                    "so saving automatically has been disabled.",
                    new string[] { "Ok" }, 0, MessageBoxIcon.Warning,
                    new AsyncCallback((result) =>
                        {
                            Guide.EndShowMessageBox(result);
                        }
                    ), null);
            }
        }
    }
    );
```

When the event is fired, make sure that you already have a device selected and whether it is still connected via the `IsConnected` property. If it is no longer connected, set your device to null (because your code already handles that scenario) and inform the user that you are turning off automatically saving. You can handle this in other ways; for example, you can allow the user to choose a new device instead. The user should never have an unexpected loss of data though, so at a minimum you need to warn him or her, which is what this example did.

The `BeginShowSelector` method actually has four overloads as well. There are two pairs of similar overloads; one in each pair takes a `PlayerIndex` to specify which user selects the device, while the other does not. In the overload used in this example, any user can select the device when the Guide pops up, and the data is stored in such a way that any user on the console can see the data. If you use the similar overload that included the `PlayerIndex` as the first parameter, a few things happen.

First, only the user who matches the player index is allowed to select the device. This can easily lead your players to consider your game has locked up if you aren't careful. For example, if you always assume that the first player is signed in and use `PlayerIndex.One` as the parameter to this method, but the only player signed in to the console is player two, he or she is never able to get out of this dialog without signing in to the first player.

> **Note**
>
> Virtually all games have a Push Start to Begin screen to detect which player is actually controlling the game.

When using a `PlayerIndex`, only the player can see the data. Much like how each game's data is segregated from any other game's data, all of the player data is separate as well.

The other two overloads each contain two new integer parameters: `sizeInBytes` and `directoryCount`. The first one is the total size of the data you plan on writing to the device, and the second is how many directories you will create. If you know this data before you create the device, using these parameters enable the user to select a device that doesn't have enough free space on it. If you do not use these parameters, the system lets you select any device.

The `StorageDevice` also has a few instance members. It has three properties, including the `IsConnected` property you saw previously (which as the name implies, tells you if it's connected). You can also use the `TotalSpace` property and `FreeSpace` property to get information about how much data the device can hold. It also has a `DeleteContainer` method, which takes in the name you passed in when opening it.

As mentioned earlier, the `StorageContainer` object is similar to the `IsolatedStorageFile` object, and they contain basically the same methods, so you can look back at the `IsolatedStorage` section to see those if you like. The container does contain two properties that the isolated storage file does not, namely the read-only `DisplayName`, which is what you used to create it with, as well as the `StorageDevice` that was used to create it.

Loading Loose Files from Your Project

One last thing you can do with storage is to load random loose files that you might need for your games. Although it's probably a better idea to use the Content Pipeline for the majority of things, there are times when using a loose file directly is useful. To see an example of this, add a new text file to your game project (do not use your content project) called data.txt. Add the following data to the file:

```
20 30
10 20
321 123
401 104
333 333
17 412
```

To make sure that the properties for this file are set correctly so it is deployed with your application, select the file in your project and view its properties. Ensure that the `Build Action` property is set to `Content`, and the `Copy to Output Directory` property is set to `Copy if Newer`. You use this data to create a new set of points to draw, which of course needs another variable to keep track of this new data:

```
List<Vector2> contentPoints;
```

Next, to initialize the variable and read the data, add the following to the end of your `Initialize` method:

```
// Read from the loose file
contentPoints = new List<Vector2>();
using (StreamReader reader = new StreamReader(
    TitleContainer.OpenStream("data.txt")))
{
    while (!reader.EndOfStream)
    {
        string currentLine = reader.ReadLine();
        string[] numbers = currentLine.Split(' ');
        contentPoints.Add(new Vector2(float.Parse(numbers[0]),
            float.Parse(numbers[1])));
    }
}
```

The important part of the snippet is the `TitleContainer.OpenStream` method. It opens loose files that you've included in your project and returns a stream containing the file's data. After you had that stream, simply read each line individually, split the line into i the two numbers, and add them to your new collection.

> **Note**
>
> The stream that is returned is always read-only.

Finally, update your `Draw` overload to draw the new set of points. You can use a different color so you know which ones are read from the file. Add the folwoing code to your `Draw` method before the `End` call to `spriteBatch`:

```
foreach (Vector2 position in contentPoints)
{
    spriteBatch.DrawString(font, "x", position, Color.Black);
}
```

This is quick and easy loading and reading a loose file. With that, you learned the basics for implementing datai storage within your games.

Summary

Storage is something gamers simply take for granted, and is many times forgotten by the developer, but its wise use is absolutely crucial to having a good experience. In this chapter, you learned the basics for using the storage APIs to save and restore state. You also learned about the nuances the different flavors of storage APIs available bring to the table.

The next chapter looks at some Xbox LIVE services available to XNA games.

15

Gamer Services

What exactly is in the mystical namespace called Gamer Services? At the most basic level, it is system services provided either by the platform itself or by the Xbox LIVE system. This is why you see things such as the `Guide` class in this chapter (provided by the system), as well as profile information and other various Xbox LIVE features. In this chapter, you will learn about:

- The Guide Class
- Gamers
- Profiles

Unfortunately, more than any other chapter in this book (that isn't platform-specific), this chapter has the most platform differences. When needed, platform differences are mentioned during the course of the chapter.

GamerServicesComponent

One of the first platform differences is the requirement (or lack thereof) of having a `GamerServicesComponent` in your application. Well, that's not entirely true—a `GamerServicesComponent` is never required, but the tasks it performs are required on some platforms.

This component does two major things that you could also do on your own, but this just simplifies it some. First, it calls `GamerServicesDispatcher.Initialize`, which is required before calling the other method it automatically calls for you, `GamerServicesDispatcher.Update`. On Xbox 360 and the Windows platforms, you must call both `Initialize` at the start of your app, as well as `Update` every frame for all Gamer Services functionality to work, including the `Guide` class. On the Windows Phone 7 platform, you do not have to call these methods to use the `Guide` class, but you do have to use the other components in the Gamer Services namespace. This is important as you find out later.

Now, the easiest way to have the correct methods called for you without you having to think about it is to add a `GamerServicesComponent` to your game's `Components` list.

This is commonly done by adding a new component to your components collection during initialization:

```
Components.Add(new GamerServicesComponent(this));
```

If you're creating an Xbox 360 or a Windows game, add a component similar to this to your game if you're using Gamer Services. Do not add this component if you are using Windows Phone 7 and using only the `Guide` class.

Guide Class

If you're tackling this book in the order the chapters are in, you've already used the `Guide` a few times. In Chapter 12, "Adding Interactivity with User Input," you used the onscreen keyboard API from the `Guide`. You also used the message box API briefly in the previous chapter, although without much explanation. For now, let's go through each of the APIs that work across all the platforms XNA supports. First, create a Windows Phone 7 game project.

Trial Mode

Trial mode is, in short, awesome, and it is required if you plan on selling your game. A well-designed trial mode can be the difference between someone buying your game and not. Far too many developers don't take enough time to make a good trial experience and their game suffers for it.

The default trial mode experience is different depending on which platform you target. For example, in order to implement a trial mode for a game you plan on publishing via Xbox LIVE Indie Games on Xbox 360, you don't have to do anything, and the system handles everything for you. Granted, it won't necessarily be a very good trial, but it will be a trial mode nonetheless! Conversely, the system does essentially nothing for you for a Windows Phone 7 project, aside from notifying you when you are and are not in trial mode. This means that for a good trial mode (and upsell opportunities), you need to implement them yourselves.

Of course, being in trial mode also means different things to the system as well. Xbox LIVE Indie Games cannot use multiplayer features in trial mode, but there is nothing stopping a Windows Phone 7 game from doing so.

So after all that, what is trial mode exactly anyway? It is the mode that your game is running in when the user has not purchased the game yet. Both of the currently available publishing pipelines (for Windows Phone 7 and Xbox 360) require you to have a trial version of the game (that is free to play) so users have an idea of what the game is like and if they want to play it more, or better yet, to buy it.

The major goal of any trial mode is to convert people who are playing your game from a free customer to a paid customer. You need to convince them that what they are playing is so great that they need to pay whatever price you deem your creation worth. Luckily, you have a lot of latitude in defining that.

One of the first things you need to know is whether or not you are actually running in trial mode. This is easily determined by looking at the `Guide.IsTrial` API call. In a published game, the API returns true if the game is currently running in trial mode (for example, has not been purchased). It returns false if it is not running in trial mode (for example, it has been purchased). This is pretty cut and dry, and for all published games, you can be assured it is correct and react accordingly.

> **Note**
>
> IsTrial can take up to 60ms to return on Windows Phone 7 projects, so do not call this every frame on those types of projects.

One of the problem's you run into during development is how you can actually test that your trial code is working correctly. Again, the differing underlying systems provide different levels of support for testing trial modes. For example, on the Xbox 360, after you deploy your game, you can launch it in trial mode for testing via the system UI, like you see in Figure 15.1.

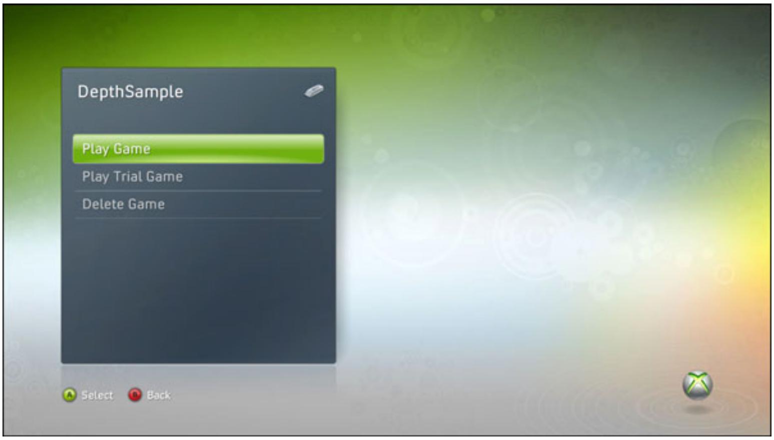


Figure 15.1 Picking startup mode for your game on Xbox 360 during development

However, on Windows Phone 7, there is no way to launch the game in trial mode via the system UI, and even if there were, there isn't a way to actually debug your code while trial mode is running (a problem shared by Xbox). During debugging, the default state of the game is considered unlocked because the majority of your game code and work is done for the case where trial mode is set to false.

Knowing this, there is an added property on the `Guide` class to help with this, named `SimulateTrialMode`. In order to turn on trial simulation, this property needs to set before the first `Update` call. In the new project, add the following line in the game's constructor:

```
Guide.SimulateTrialMode = true;
```

This tells the runtime that you want to simulate the trial mode, which forces a game that is not in trial mode to return true from the `IsTrial` property. This ensures that `IsTrial` returns true until you either turn off via `SimulateTrialMode`, or when you simulate purchasing of the game.

Note

After a game's trial status has gone from `IsTrial` being true to `IsTrial` being false, from trial simulation, purchasing, or any other mechanism, it never returns to the state where `IsTrial` returns true unless the application is exited and restarted.

You're probably thinking to yourself, wait a minute, simulate purchasing of the game? As mentioned earlier, the best trial modes are an introduction to the game with the hope of getting your customer to purchase it. This implies that you need a way to enable the user to actually do just that. To show how this works, let's simulate it via one of the gestures in the touch class, so add the following to your game's constructor as well:

```
TouchPanel.EnabledGestures = GestureType.Tap | GestureType.Hold;
```

You use only one of these for now, but add them all now so you don't have to change it later. Now, add the following code to your `Update` method to handle the gestures:

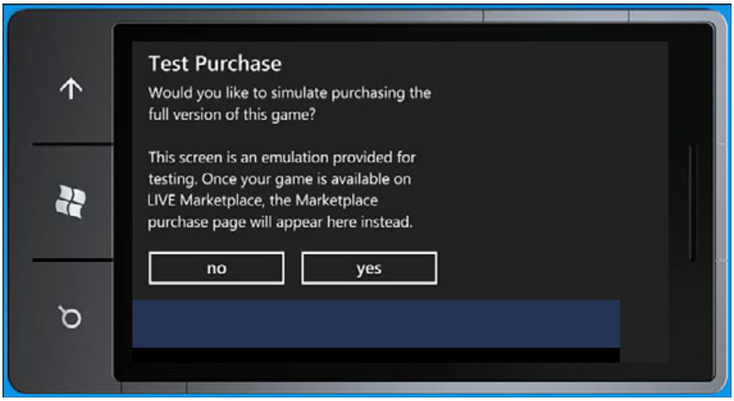
```
while (TouchPanel.IsGestureAvailable)
{
    GestureSample gesture = TouchPanel.ReadGesture();
    switch (gesture.GestureType)
    {
        case GestureType.Hold:
            if (Guide.IsTrialMode)
            {
                Guide.ShowMarketplace(PlayerIndex.One);
            }
            break;
    }
}
```

The important piece here is the `Guide.ShowMarketplace` call, which is where you have the opportunity to sell your game. For a game that has already been released, this brings up the system user interface to enable customers to purchase your game. If they do so, they switch immediately to the purchased version (`IsTrial` returns false). In debugging builds, this call instead shows a simple message asking if you would like to simulate the purchase of the game, much like you see in Figure 15.2.

Note

The check for `IsTrial` here isn't necessary, because the `ShowMarketplace` call silently no-ops if you are not in trial mode; it was added for completion sake.

The `ShowMarketplace` call is intended to be a fire and forget it type of method. You don't need to check its return (it has none), and the only exception it will ever throw is if you pass an invalid value to its parameter. For Windows Phone 7 projects, the parameter should always be `PlayerIndex.One`, and for Xbox projects, it should be the index of the player who is initiating the purchase screen. Depending on what happens during the call, the value of `IsTrial` might change, and that is what your game should key on.



Test Purchase

Would you like to simulate purchasing the full version of this game?

This screen is an emulation provided for testing. Once your game is available on LIVE Marketplace, the Marketplace purchase page will appear here instead.

Figure 15.2 Simulate purchasing of a game

An example of a good trial mode with upsell is a simple game with multiple levels. You can allow players to play the first five levels in the trial mode and when they finish those levels, bring them to a new screen saying that they could continue the game right now by purchasing the full game and using the `ShowMarketplace` API to take them there.

As was mentioned earlier, the system handles everything (including upsell) for your game on Xbox 360. There is a strict time limit on all trial games (exactly 8 minutes as of this writing, but is subject to change at any time), and when the time limit expires a new screen appears on top of the game enabling users to continue if they purchase the game, or to exit the game. This screen obscures the majority of the game behind it (the entire title safe area, plus some), and no input is routed to the game when that screen is up. The `Guide.IsVisible` property also begins to return true.

This brings us to the next property available on the Guide, `IsVisible`. This returns true if the guide is currently visible (for example, your game is obscured), and false otherwise. For example, change the `Clear` call in your `Draw` method to the following:

```
GraphicsDevice.Clear(Guide.IsVisible ? Color.Black : Color.CornflowerBlue);
```

Now, if you run your application and hold the screen down to make the gesture fire, when the simulate purchase screen is up, the entire screen turns black, rather than the faded `CornflowerBlue`. The most common usage of this property is to ensure you don't attempt to load another dialog when one is already obscuring your game, which you see in a few moments.

The last property we talk about in this section is the `IsScreenSaverEnabled` property, which you can add to your project in the constructor:

```
Guide.IsScreenSaverEnabled = false;
```

The name of the property is a bit misleading, but what it does is quite simple. It enables (or disable) the system from thinking the user has stopped playing it. On the Xbox

360, this manifests itself by dimming the screen; on Windows Phone 7, it manifests itself by engaging the lock screen. If you had an area of the game where you expect long periods of time without user interaction (for example, playing a very long cut scene on Xbox, or a game that is controlled completely by the accelerometer on the phone), you should set this property to false so the system doesn't kick in and dim the screen or lock it. You shouldn't carelessly set this property to false all the time though.

> **Note**
>
> The `IsScreenSaverEnabled` property is application specific, not system wide, so even if you set it to false and exit the game, it reverts back to the system setting after the application quits.

The last method that we discuss in this section is a common way to show simple messages. The marketplace simulated purchase uses this API behind the scenes, and you used it in previous chapters as well. As you probably surmised, this API is `ShowMessageBox`. This API follows the standard .NET async pattern, so it is actually split into two separate APIs, `BeginShowMessageBox` and `EndShowMessageBox`. Add a new case statement into the gesture code in your `Update` method:

```
case GestureType.Tap:
    if (!Guide.IsVisible)
    {
        Guide.BeginShowMessageBox("Testing", "This is a test message box.",
            new string[] { "ok", "cancel" }, 0, MessageBoxIcon.None,
            new AsyncCallback((result) =>
                {
                    int? button = Guide.EndShowMessageBox(result);
                }), null);
    }
    break;
```

The first parameter to the `BeginShowMessageBox` API is the title of the message box, followed by the actual text of the message to be displayed. Next is an array of strings that represent the text that the buttons for the message box will have. For Xbox, this array must have at least one member in it, and up to a maximum of three, and on Windows Phone, it needs to have either one or two members. After the buttons are listed, the next parameter controls which button is the default selected button. This parameter has no effect on Windows Phone projects. Lastly, is the `MessageBoxIcon` parameter that controls the icon the message box displays on Xbox, and the sound it makes on Windows Phone.

After the async call finishes, call the `EndShowMessageBox` method to get the button that was pressed. It returns the index of the button name or null if the message box was cancelled (by either the B button on Xbox or the Back button on Windows Phone).

This overload of `BeginShowMessageBox` enables anyone to use the dialog. However, on Xbox 360, if you want a specific player to be the only one who can respond to the dialog, use the other overload that includes the `PlayerIndex` parameter and pass the index of the player you want to control the message box in the parameter.

Now the Bad News

Some of the functions in the `Guide` class are not available on Windows Phone 7 because they have no equivalent system UI, so it makes a bit of sense for them to be missing. However, a few functions in the `Guide` and everything else in Gamer Services (and this chapter) do not work on Windows Phone 7 unless you are a partner Xbox LIVE game. If you don't know what that means, then unfortunately you are probably not one. The API calls still exist on the platform and work if you have a partnership agreement with Microsoft, but if you do not, you get an exception trying to initialize the `GamerServicesDispatcher` (even if via `GamerServicesComponent`). This is the reason that the `Guide` methods on the phone do not require these components.

Platform-Specific Guide Functionality

Although the rest of these APIs work on Windows, they're there only to help debug Xbox functionality. So for the rest of this chapter, you deal with Xbox projects. First, create a new Xbox Game project.

If you remember from earlier in this chapter, you need to initialize Gamer Services before doing anything else on Xbox. So, add the following to your `Initialize` overload:

```
Components.Add(new GamerServicesComponent(this));
```

Notifications

As you move on to platform-specific features, a good place to start is notifications. You've probably seen quite a few notifications pop up while you were playing on your Xbox, from when you first sign in to Xbox LIVE, to when you get an achievement, to when you see your friends logging on. They're ubiquitous on the platform! The XNA runtime includes two separate APIs to control the behavior of these notifications as well.

The first is the `Guide.DelayNotifications` method. This takes a single parameter that is a `TimeSpan` for how long you want to delay the notifications, up to a maximum of 120 seconds (two minutes). If you pass in a larger delay, the maximum is used. This method ensures that key portions of your game (for example, a cut scene) are not interrupted by the notifications. After the delay is over, the system delivers the notifications in the order they were received.

> **Note**
>
> You cannot call `DelayNotifications` continuously. The system needs time to deliver notifications.

The other is a property called `Guide.NotificationPosition`, which dictates where the notification shows up on the screen. There are nine different areas where the notifications can show up: top left, center or right; bottom left, center, or right; or middle left, center, or right. The default location is the bottom center, but you are free to move it elsewhere.

Other Players

One of the great features of Xbox LIVE is the capability to interact with your friends, even if you aren't actually playing a game with them at the time. To update your new game project to show off some of these features, add a new variable to your game:

```
GamePadState lastState;
```

Use this to easily detect button presses. You should also add a quick helper method to check for the presses, so you don't have to repeat the same code pattern multiple times:

```
/// <summary>
/// Simple helper to determine if a button was pressed
/// </summary>
public bool WasButtonPressed(Buttons button, GamePadState current, GamePadState
➥last)
{
    return (current.IsButtonUp(button) && last.IsButtonDown(button));
}
```

Now, add the following code to your `Update` method to see the next system interface call:

```
GamePadState state = GamePad.GetState(PlayerIndex.One);
if (!Guide.IsVisible)
{
    if (WasButtonPressed(Buttons.A, state, lastState))
    {
        Guide.ShowFriends(PlayerIndex.One);
    }
}
lastState = state;
```

> **Note**
>
> Again, in this example, and the rest of the examples in this chapter, you use `PlayerIndex.One` as your player. In a real game, ensure that this parameter is a valid player index.

With this code, when you press and release the A button on your controller, it shows the system user interface for your friends, as in Figure 15.3. Use this in a game menu to show your friends.

You can also ask someone new to be your friend by using a call such as the following:

```
Guide.ShowFriendRequest(PlayerIndex.One, otherPlayer);
```

This is where the `otherPlayer` parameter is of type Gamer (which is discussed later in this chapter). This brings up the system user interface to enable the user to send a friend request to the other person specified by `otherPlayer`. Because this is the system UI providing this, all features of requesting a friend are here, including attaching voice and text messages. When your game has individuals playing a multiplayer game in a public session, it gives your players an opportunity to request to be friends with the people they enjoyed playing with.

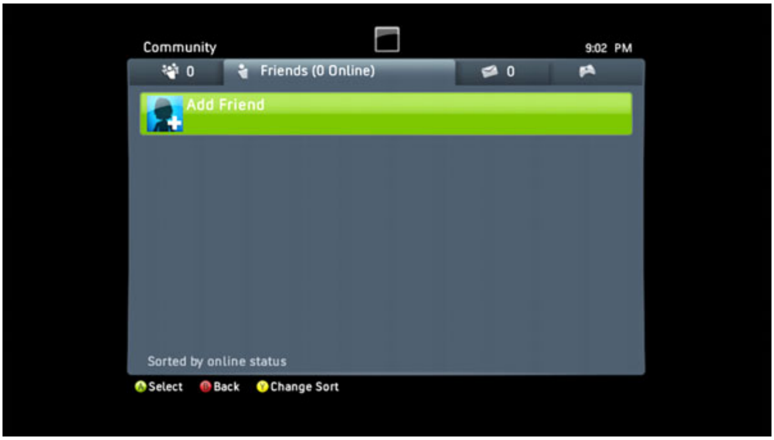


Figure 15.3 The system's friend user interface

A similar system screen to the friends is the list of players you've recently played online with. You can add support for this screen by adding the following to your `Update` method before storing the previous state:

```
if (WasButtonPressed(Buttons.X, state, lastState))
{
    Guide.ShowPlayers(PlayerIndex.One);
}
```

Of course, what if you play with someone who was a total jerk, and saying inappropriate things, and generally being a bad person. Xbox LIVE enables a player to submit feedback on other players. If you show a list of players within your game, you might want to give your players a chance to submit feedback on others. You can do so with the following API:

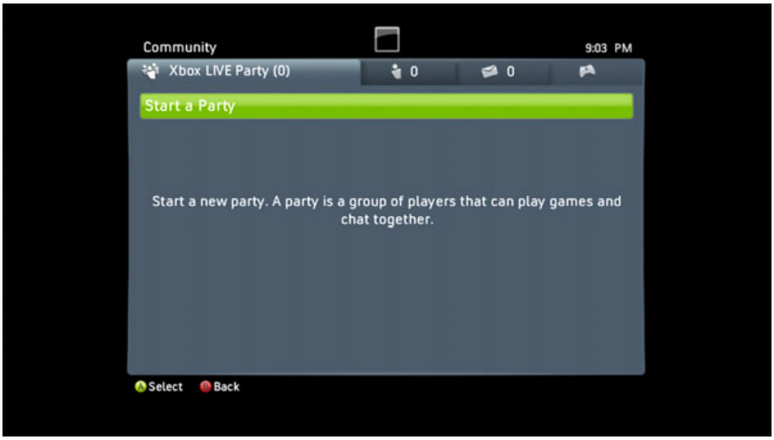
```
Guide.ShowPlayerReview(PlayerIndex.One, otherPlayer);
```

This API brings up the system screen for submitting player review, and like the earlier example uses a parameter of type `Gamer`, which is discussed shortly in this chapter. Perhaps you don't want to submit feedback on a player; you just want to see more information about him or her. Then you use the following API to enable you to see his or her gamer card:

```
Guide.ShowGamerCard(PlayerIndex.One, otherPlayer);
```

One of the more recent features with Xbox LIVE is the concept of parties, in which you can be in a group (or party) of people where you can chat and interact, even if you are all playing different games. You can add the following call to your `Update` method to see the party screen as shown in Figure 15.4:

```
if (WasButtonPressed(Buttons.B, state, lastState))
{
    Guide.ShowParty(PlayerIndex.One);
}
```



Start a new party. A party is a group of players that can play games and chat together.

Figure 15.4 The system's party user interface

You can also see the party session system screen, which enables you to invite your party members to your games by using the following API:

```
Guide.ShowPartySessions(PlayerIndex.One, otherPlayer);
```

There are also other ways to invite other players to your party later this chapter. You can also use one of the guide methods to invite other players to your current session (which you learn more about in the next chapter), but the API call is the following:

```
Guide.ShowGameInvite(PlayerIndex.One, listOfPlayers);
```

> **Note**
>
> There is another overload for `ShowGameInvite` that takes a string that is the session identification for use in mobile games if you happen to be an Xbox LIVE partner.

This sends an invite to your current session to all players in the list you passed in. You learn about sessions and inviting and playing in them in the next chapter.

Messaging and Signing In

Unlike the message box API discussed earlier, which simply shows a message on your local screen, there are two other APIs that enable you to send messages to other players over the

Xbox LIVE service. To see what messages you currently have, add this code to your project in the `Update` method:

```
if (WasButtonPressed(Buttons.Y, state, lastState))
{
    Guide.ShowMessages(PlayerIndex.One);
}
```

This pops up the system screen with your messages as seen in Figure 15.5. You can also use the following API to show the system screen enabling you to send a message to someone else:

```
Guide.ShowComposeMessage(PlayerIndex.One, "Hello", listOfPlayers);
```

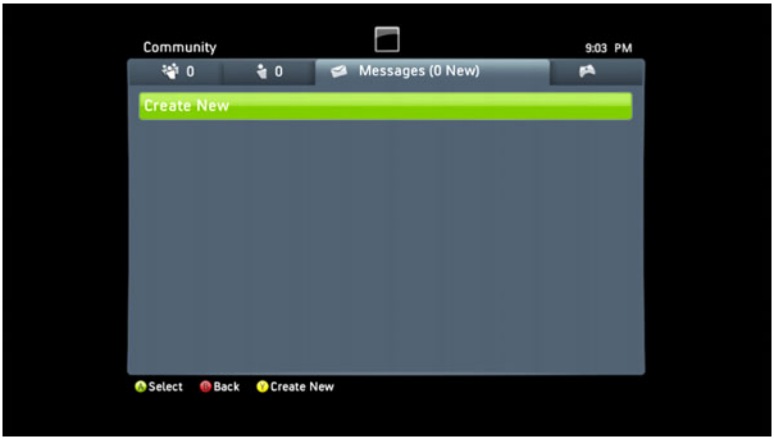


Figure 15.5 The system's message user interface

The text you pass into the method pre-populates the message (and it must be fewer than 200 characters). You can also pass in null as the list of players to have the system show the select gamertag screen before the message is sent.

Lastly, what if you detect that no one is signed in, or not enough people are signed in for your game (you learn how to determine this in the next section)? To tell people to sign in to the game, add the following code to your `Update` method:

```
if (WasButtonPressed(Buttons.LeftShoulder, state, lastState))
{
    Guide.ShowSignIn(4, false);
}
```

This shows the sign-in system screen. The first pane tells the system how many panes to show for the sign in, and valid values are 1, 2, and 4. With a value of 1, a single sign-in pane shows up for one person to sign in. With a value of 2, you have two panes side by side enabling players to sign in. With four, you have a square of panes available for signing in. The next parameter determines which type of profiles can sign in. If the value is true, only profiles that are online enabled can sign in. If it is false, everyone can sign in. If it is true, local players can still sign in as guests of an online account. With these parameters, the screen appears as it does in Figure 15.6.

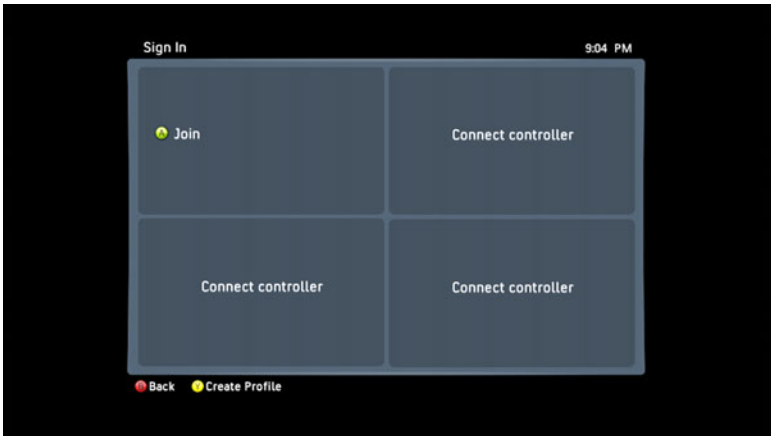


Figure 15.6 The system's sign-in user interface

Gamers and Profiles

Now it's time to move on to gamers and profiles. Rather than starting an entire new project, let's just remove the four checks to the `WasButtonPressed` method in your `Update` overload. You should also add a new `SpriteFont` object called font to your Content project because you draw text in a few moments. Then add the following variables:

```
SpriteFont font;
Gamer lastGamer;
GamerProfile lastGamerProfile;
Texture2D gamerPicture;
```

Load the font in `LoadContent` as usual:

```
font = Content.Load<SpriteFont>("font");
```

Now, you need to do something with the players that are currently signed in, so first add the following code to your `Initialize` method to set all the rest of your variables:

```
SignedInGamer.SignedIn += new EventHandler<SignedInEventArgs>((o,e) =>
{
    lastGamer = e.Gamer;
    lastGamer.BeginGetProfile(new AsyncCallback( (result) =>
        {
            lastGamerProfile =
                lastGamer.EndGetProfile(result);
            gamerPicture = Texture2D.FromStream(
                GraphicsDevice, lastGamerProfile.GetGamerPicture());
        }), null);
});
```

The `SignedInGamer` class has two static events to detect when players are coming and going, namely the `SignedIn` and `SignedOut` events. In this example, you hook the `SignedIn` event and store the gamer that signed in to your variable. Each gamer also has an associated profile, which you store here using the `async` pattern call. There is also a synchronous `GetProfile` method you can call instead, but using the `async` pattern is the correct way to retrieve the profile. Lastly, store the profile's gamer picture into a newly created Texture by using the `GetGamerPicture` method, which returns a stream.

Note

Anyone who is signed in when the game launches receives a `SignedIn` event fired. You do not need to special case startup.

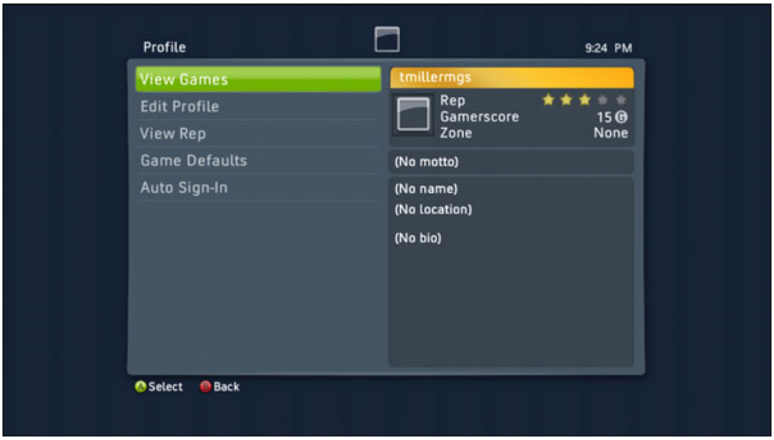
Next, replace your `Draw` overload with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    if (gamerPicture != null)
    {
        spriteBatch.Draw(gamerPicture, new Vector2(400,100), Color.White);
    }
    if (lastGamer != null)
    {
        Vector2 pos = new Vector2(50,30);
        spriteBatch.DrawString(font, lastGamer.Gamertag, pos, Color.White);
        if (lastGamerProfile != null)
        {
            pos.Y += 20;
            spriteBatch.DrawString(font, lastGamerProfile.GamerScore.ToString(),
                pos, Color.White);
        }
    }
    spriteBatch.End();
}
```

Here you simply use each of the variables you defined so far to display some information. First, draw the gamer picture onto the screen if the texture has been created, then draw the gamer tag of the profile after the gamer is stored, and finally draw the gamer score from the profile. Because the `SignedIn` event is hooked and storing the data for anyone that logs in, if you log in someone new while this example is running, the data onscreen changes to the new player.

You can also use the gamer in earlier `Guide` methods that required a gamer class. As an example, adding the following code to your `Update` method shows the gamer card of the signed-in profile as seen in Figure 15.7:

```
if (WasButtonPressed(Buttons.A, state, lastState))
{
    if (lastGamer != null)
    {
        Guide.ShowGamerCard(PlayerIndex.One, lastGamer);
    }
}
```



tmillermgs
Rep ★★★
Gamerscore 15
Zone None
(No motto)
(No name)
(No location)
(No bio)

Figure 15.7 Showing a user's gamer card

Before moving on, let's take a moment to look at the other pieces of information stored in these objects. The `Gamer` object is an abstract class, so it has a few derived classes, but it only has a couple extra members directly on itself. It has the `DisplayName` property, which many times is simply the `GamerTag`, but can be something different. It also includes the `Tag` property for you to do with what you want. Finally, it has the unsupported `LeaderboardWriter` property. This property is available only for Xbox LIVE partners.

The `GamerProfile` object has quite a few things you haven't looked at yet though, mostly around portions of the players' profiles (imagine that). It has the `GamerZone` property, which can be `Family`, `Pro`, `Recreation`, `Underground`, or `Unknown`. You can also get the `Motto` property to view a player's motto, which can be empty if he or she hasn't set it. You can see the player's `Reputation` by checking that property, which is a float value between 0.0f and 5.0f representing the number of stars of reputation that player has. The float values enable you to have fractional stars as well.

You can also use the `TitlesPlayed` property to determine how many games a player actually played, along with `TotalAchievements` to see how many achievements he or she earned. Finally, you can use the `Region` property to determine what region a player is located. This covers all the basic properties for these objects.

> **Note**
>
> In this example, you use the event to display data about the last person who was signed in; however, there is a static collection of `SignedInGamer` called `SignedInGamers` that enable you to view the collection of gamers currently signed in as well.

Remember that the `Gamer` class is abstract, which means that the object you are currently using is one of the derived classes; namely, the `SignedInGamer` object itself. This class has much more data than the abstract `Gamer`, so update your variable to declare it of the following type instead:

```
SignedInGamer lastGamer;
```

One of the things on this object are several methods dealing with achievements. Unfortunately, these methods are available only for partner Xbox LIVE games. There are plenty of other members you can actually use though.

For one, you can detect whether players are signed in to live via the `IsSignedInToLive` property. This is always false for local accounts, but it can be false for Xbox LIVE accounts as well if they aren't actually connected to Xbox LIVE. You can also see which controller is paired with this gamer by checking the `PlayerIndex` property.

You can detect whether gamers are in a party by checking the `PartySize` property. If it is greater than zero, then they are in a party. If players are guests of a signed-in Xbox LIVE account, then the `IsGuest` property returns true. If you need to know whether a certain microphone is the headset for one of your gamers, you can use the `IsHeadSet` property to detect that as well!

GameDefaults

A feature often overlooked on Xbox is the fact that gamers can set up a list of defaults for various options in games that they play. Good games know which of these options make sense for them and set the default for a player to what they request. This is found in the `GameDefaults` object of the `GameDefaults` property.

There are quite a few Boolean properties available to check including `AccelerateWithButtons`, `AutoAim`, `AutoCenter`, `BrakeWithButtons`, `InvertYAxis`, `ManualTransmision`, and `MoveWithRightThumbStick`. For example, in a racing game with both an automatic and manual transmission, the default for a gamer is automatic if `ManualTransmision` is false, and manual transmission otherwise.

The other defaults are more than Booleans. You have both `PrimaryColor` and `SecondaryColor`, which are returned as colors (or null). The `ControllerSensitivity` property is an enumeration that is either `High`, `Medium`, or `Low`. Similarly, the `GameDifficulty` property is an enumeration that can be `Hard`, `Normal`, or `Easy`. Lastly, the `RacingCameraAngle` is an enumeration that can be `Back`, `Front`, or `Inside`.

Your game needs to be aware of these properties and set the initial values for a gamer to respect him or her.

Presence

The `Presence` property is one of the more interesting things you can use. This property enables you to set a well-known string that anyone can see to let others know where a person is in game play. The property itself is a `GamerPresence` object that has two properties, `PresenceMode` that is an enumeration of a series of predefined strings, and `PresenceValue` that is extra data used for some of the strings. For example, make the following call in your code:

```
lastGamer.Presence.PresenceMode = GamerPresenceMode.OnARoll;
```

Now, when someone else looks you up online (via http://xbox.com or Xbox 360), your profile includes the following:

```
XNA Creators Club
creators.xna.com
On a roll
```

If this game is published on Xbox LIVE Indie Games, the first line changes to "Xbox LIVE Indie Games" and the second is your game's title. The third line is always the presence string specified by the `PresenceMode` property. Some of the values of `PresenceMode` enable you to add extra data; namely, `Stage`, `Level`, `Score`, `CoopStage`, `CoopLevel`, `VersusScore`. This enables you to have the presence string say something like "Playing Stage 3." Although updating your player's presence isn't a requirement, it certainly adds to the polish of your game and is highly recommended.

Privileges

Proper handling of the `Privileges` property is required. These represent the various things gamers may or may not be able to do. Perhaps they are `Silver` subscribers, or perhaps they are children whose parents have set up restrictions on their ability to play. Regardless of the reason, you should always obey the privileges. If you don't, the system still disallows the action, but it can be a very poor user experience, or worse, a crash.

Privileges come in one of two forms. They either return a Boolean, in which case the privilege is either allowed or disallowed, or they can return a `GamerPrivilegeSetting`, which has three choices: `Blocked` (they have no access to this feature), `Everyone` (they have full access to this feature), or `FriendsOnly` (they have access to this feature only with friends).

The first privilege is `AllowOnlineCommunication`, which dictates whether a gamer can have any type of online communication (text, chat, video, and so on). Check this privilege before allowing someone to communicate with someone else (such as with the `Guide.ShowComposeMessage` method).

Next is `AllowOnlineSessions`, which essentially dictates whether a gamer can play multiplayer games. A good example of using this privilege is to either not show the multiplayer option or show it but have it disabled in your game. A best practice is to include some status text informing the player why the multiplayer is disabled.

After that is `AllowPremiumContent`, which is mainly used for Xbox LIVE partner games. If you want to restrict your game based on this, you can. The `AllowProfileViewing` privilege dictates whether the gamer is allowed to view other players' profiles, so you do not allow them to call `Guide.ShowGamerCard`, for example, if this privilege is not available.

Next is `AllowPurchaseContent`, which dictates whether the gamer is allowed to spend money to purchase content. This is another place where some best practices can go a long way. If the gamer is not allowed to purchase, don't give him or her the option, disable the option, and don't call `Guide.ShowMarketPlace`.

The last two privileges are `AllowTradeContent` and `AllowUserGeneratedContent`. These two can be tricky. For example, if your game enables the creation of levels, and you can share those levels, that is both user-generated content and trading content. This is extremely difficult for the system to detect (and impossible in many cases), so ensure that your game handles these correctly.

With Friends Like This...

Because some of the privileges require you to know whether someone is a friend in order to determine whether an action is allowed, there must be an easy way to figure this out. Of course, there is! Use the `IsFriend` method, which returns true if the gamer you pass to it is your friend and false otherwise.

What if you want to see all of your friends though? Well, use the `GetFriends` method, which returns to you a collection of all your friends. However, because it is possible that some of your friends are not signed in locally, the `SignedInGamer` object doesn't work for your collection of friends. Instead, it is a collection of `FriendGamer`, which has quite a few different members.

You can check the status of your friends via a number of different properties such as `IsOnline` (to check whether they're actually signed in), `IsAway` (perhaps they fell asleep playing a game), `IsPlaying` (to check whether they're playing a game), `IsBusy` (just in case they're busy), and `IsJoinable` (to check whether they're playing a game that can be joined).

You can use the FriendRequestReceivedFrom property to determine whether you received a friend request (it returns true if you have, and false otherwise). Conversely, you can use the `FriendRequestSentTo` property to see whether you have sent a friend request.

Just like you can set your presence, you can also see your friend's `Presence`, although it is returned as the actual string because it can be from any game, including games that are not XNA games. You can also use the `HasVoice` property to determine whether your friend has the voice capability.

Lastly, if you sent an invite to your friend (which you learn more about in the next chapter), you can get status of that as well. If he or she accepts the invite, the `InviteAccepted` property returns true. If your friend rejects it, the `InviteRejectedProperty` returns true (and `InviteAccepted` remains false). You can also use the `InviteSentTo` property to determine whether you actually sent an invite to that friend. It returns true if you have, and false otherwise. You can also see whether 'he or she invited you via the `InviteReceivedFrom` property, which returns true if he or she sent you an invite, and false otherwise.

Summary

Although the term Gamer Services is a bit of a misnomer, there is quite a bit of great information here. You learned about the system user experiences available, as well as how to interact with many of the Xbox LIVE services and information about the profile's signed in to your console.

In the next chapter, you can learn how to put it all to use by creating some multiplayer networking code.

16

Multiplayer Networking

Single player games can provide a very rewarding story experience, but nothing brings out competition more than playing a game against other real human beings from across the world. Some of the top selling games of all time offer compelling multiplayer experiences that extend the amount of time players continue to come back and replay their game.

In this chapter you will learn about:

- Setting up for networking development
- Creating a network session
- How to find and join a network session
- Building a game lobby
- Sending player invites

XNA Game Studio 4.0 provides real-time synchronous multiplayer networking APIs for Windows and Xbox 360 that use the Xbox LIVE service.

Using `GamerServices` functionality from a Windows Phone 7 game requires that the game's author be part of the Xbox LIVE Registered Developer Program.

Multiplayer Games

Multiplayer games enable the game player to enjoy the cooperation or competition from other players. Before computer networks were popular, this was normally done by enabling multiple players to play on a single television screen. Games can enable both players to be displayed on the screen at the same time and place or can offer multiple points of view by splitting the screen into multiple areas called split screen.

Many gamers today enjoy the availability of high speed Internet access. Personal computers and gaming consoles can connect over the Internet to enable multiple players to be connected together while playing their games.

XNA Game Studio 4.0 provides the capability to create, find, and connect to network sessions between Windows PCs using Games for Windows LIVE and Xbox 360s using Xbox LIVE. The networking APIs that you will use are part of the

`Microsoft.Xna.Framework.Net` assembly and use the namespace of the same name. If your Windows or Xbox 360 project does not currently reference the `Microsoft.Xna.Framework.Net` assembly, then add the reference now. You need the reference of the samples for the remainder of this chapter. You can also add the using statement for the `Microsoft.Xna.Framework.Net` namespace to your `Game` class. Add the following line of code to your C# file that contains your `Game` class:

```
using Microsoft.Xna.Framework.Net;
```

Getting Ready for Networking Development

Building a multiplayer networked game requires that you have more than one machine that is running your game. One common development scenario is to build your game to run on both Xbox 360 and Windows. Because XNA Game Studio supports building your project for both of these platforms, you can create your game, deploy the game to your Xbox, and run the Windows version at the same time.

Running an XNA game that uses networking during development has some requirements for the gamer profile that is playing the game. On Xbox 360, the gamer profile must have an XNA Creators Club membership. Because the same membership is required to do XNA development on Xbox 360, you should already have that requirement out of the way. The other requirement on Xbox 360 is an Xbox LIVE Gold membership to connect a `Player` or `Ranked` session over the Internet. Only a Silver membership is required if you use `SystemLink`, which enables multiple machines to connect over a local area network—that is what you normally do when developing your game.

On Windows, only a local Games for Windows LIVE is required for building a game using `SystemLink`. If you use a profile that has a Silver or Gold account, you also need an XNA Creators Club membership. Because you will develop over `SystemLink`, you should not need a second XNA Creators Club membership. Use a local account or you will get an exception.

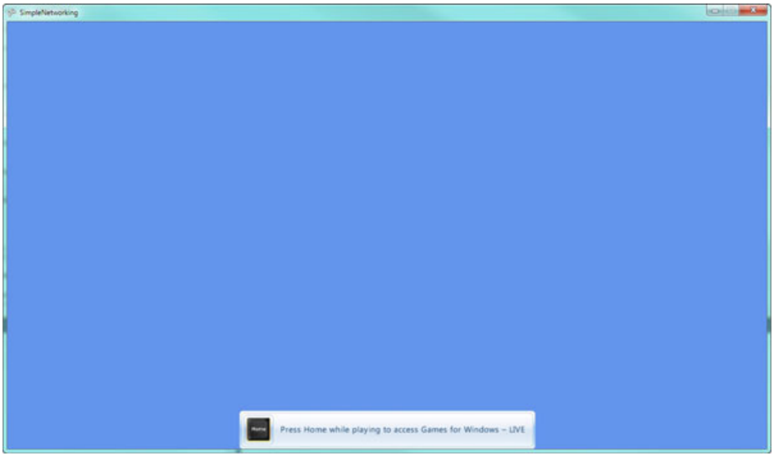
To use the networking APIs in XNA, initialize `GamerServices`. Like you did in Chapter 15, "Gamer Services," add the `GamerServicesComponent` to your game.

Add the following code to your `Game` class constructor:

```
graphics = new GraphicsDeviceManager(this);
graphics.PreferredBackBufferWidth = 1280;
graphics.PreferredBackBufferHeight = 720;
Content.RootDirectory = "Content";
// Initialize GamerServices
Components.Add(new GamerServicesComponent(this));
```

If you run the game on Windows, the game runs Game for Windows LIVE (see Figure 16.1).

Press the Home key to bring up the guide in Game for Windows LIVE (see Figure 16.2).


SimpleNetworking
Home

Press Home while playing to access Games for Windows – LIVE

Figure 16.1 Game starting Games for Windows LIVE

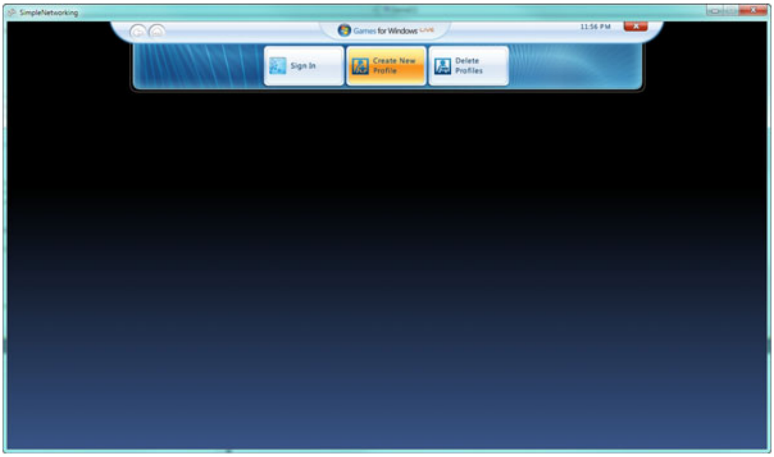

SimpleNetworking
Games for Windows LIVE
11:56 PM
Sign In
Create New Profile
Delete Profiles

Figure 16.2 Games for Windows LIVE guide

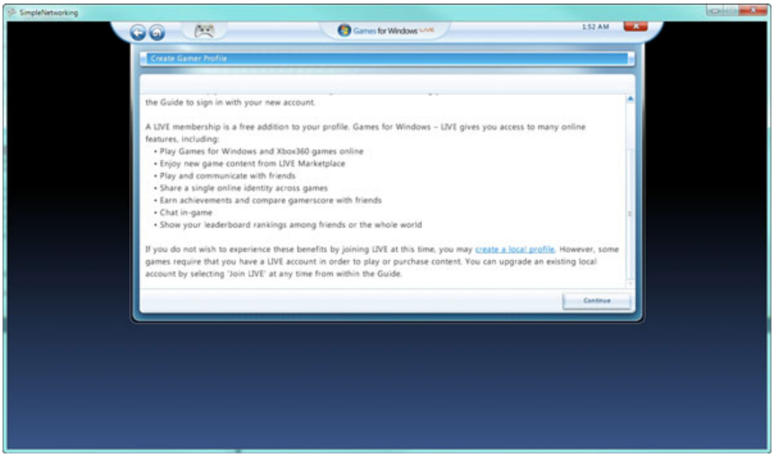
If you already have a profile signed, sign the profile out so that your screen looks like Figure 16.2.

Now, create a local profile. To do this, select Create New Profile located in the middle button area of the guide.

Note

You can use the keyboard, mouse, and Xbox 360 controller to interact with the Game for Windows LIVE guide. In this chapter, the samples assume you are using a wired Xbox 360 controller plugged into your PC.

After you click the Create New Profile button, scroll down to Create a Local Profile link and click the link (see Figure 16.3).



the Guide to sign in with your new account.

A LIVE membership is a free addition to your profile. Games for Windows – LIVE gives you access to many online features, including:

- Play Games for Windows and Xbox360 games online
- Enjoy new game content from LIVE Marketplace
- Play and communicate with friends
- Share a single online identity across games
- Earn achievements and compare gamerscore with friends
- Chat in-game
- Show your leaderboard rankings among friends or the whole world

If you do not wish to experience these benefits by joining LIVE at this time, you may create a local profile. However, some games require that you have a LIVE account in order to play or purchase content. You can upgrade an existing local account by selecting 'Join LIVE' at any time from within the Guide.

Figure 16.3 Creating a local profile in Games for Windows LIVE

Complete the process to create the new local profile. You can select to have that profile auto login when you start, which saves you time while you are develop the game and saves you the step of having to sign in the local profile every time you start the game.

Main Menu and State Management

The sample consists of multiple text menu screens and a game play screen that consists of moving the player's gamertag around. Depending on the current state of the game, you will want to display a different menu and update a different set of logic. To do this, use a simple enumeration that tracks the game's current state.

Outside the `Game` class but within the same file and namespace, add the following enumeration:

```
// Different states the game can be in
public enum GameState { MainMenu, CreateSession, FindSession, GameLobby,
➥PlayingGame };
```

As you can see from the `GameState` enumeration, the sample game contains five different states. As you progress through this chapter, you implement the update and display logic for each of these.

During the course of the game, you will want to display some text for a short period of time. Use this for a number of notifications including when a player joins or leaves a session. Use a new structure that contains the message and the time remaining to display the text. Add the following structure to your game:

```
// Structure used to display game messages over a TimeSpan
public struct DisplayMessage
{
    public string Message;
    public TimeSpan DisplayTime;

    public DisplayMessage(string message, TimeSpan displayTime)
    {
        Message = message;
        DisplayTime = displayTime;
    }
}
```

The `Game` class needs a few member variables. You need to hold the current state that you created an enumeration for and you need a list of the game messages to be displayed. Add the following member variables to your `Game` class:

```
// Current state our game is in we start at the main menu
GameState gameState = GameState.MainMenu;

// The messages to display on the screen
// We will use this to show events that occur like
// a player joining a session
List<DisplayMessage> gameMessages = new List<DisplayMessage>();

SpriteFont spriteFont;
GamePadState currentGamePadState;
GamePadState lastGamePadState;
Random random = new Random();
```

In addition to the `gameState` and `gameMessages`, create variables for the `SpriteFont` you will use, the `GamePadState` for the current and last frame, and a new Random generator that you will use within the game.

In your game's `LoadContent` method, load the `SpriteFont` you plan to use for the text in the sample. Create a new `SpriteFont` in the content pipeline and set the text size to 24. Add the following to the `LoadContent` method:

```
spriteFont = Content.Load<SpriteFont>("SpriteFont1");
```

Add the following code to your game's `Update` method:

```
// If there is no user signed in after 5 seconds
// then we show the sign in
if (gameTime.TotalGameTime.Seconds > 5)
{
    if (Gamer.SignedInGamers.Count == 0 &&
        !Guide.IsVisible)
        Guide.ShowSignIn(1, false);
}

// Store the current GamePadState
currentGamePadState = GamePad.GetState(PlayerIndex.One);

// Switch to determine which update method to call
// based on the current game state
switch (gameState)
{
    case GameState.MainMenu:
        MainMenuUpdate();
        break;
}

// Store the game pad state for next frame
lastGamePadState = currentGamePadState;

// Update the DisplayTime of the current display message
if (gameMessages.Count > 0)
{
    DisplayMessage currentMessage = gameMessages[0];
    currentMessage.DisplayTime -= gameTime.ElapsedGameTime;

    // Remove the message if the time is up
    if (currentMessage.DisplayTime <= TimeSpan.Zero)
    {
        gameMessages.RemoveAt(0);
    }
    else
    {
        gameMessages[0] = currentMessage;
    }
}
```

Check whether any user is currently signed in. If a user is not signed in after 5 seconds, call `Guide.ShowSignIn`, which displays the sign-in dialog in the guide. Use a switch statement to call the appropriate update method depending on the current `gameState` value. Right now, you have only the `MainMenuUpdate` method, but as you progress through the chapter, you add more for each of the different `GameState` values. The last part of the `Update` method loops over the `gameMessages` to determine whether any of them need to be removed so they are no longer displayed.

Add the `MainMenuUpdate` method that will handle user input and logic when our sample is at the main menu. Add the following method to your `Game` class:

```
// Update method for the MainMenu GameState
private void MainMenuUpdate()
{
    // Exit the game
    if (ButtonPressed(Buttons.Back))
        Exit();
}
```

The main menu update does not do much now, but later you add the capability to create and find a session. If the user presses the `Back` button, you exit the sample.

To check for a single button press in many of your menus, use a helper method that you can add to your `Game` class.

```
// Helper to determine if a button was pressed but will
// not allow repeat presses over several frames
bool ButtonPressed(Buttons button)
{
    // Don't process buttons when the guide is visible
    if (Guide.IsVisible)
        return false;
    return currentGamePadState.IsButtonDown(button) &&
            lastGamePadState.IsButtonUp(button);
}
```

The `ButtonPressed` method returns true if the button was pressed on this frame and does not allow for multiple button presses.

To call the draw methods for the different menu screens in the sample, add the following to your game's `Draw` method.

```
// Switch to call the correct draw method for the
// current game state
switch (gameState)
{
    case GameState.MainMenu:
        MainMenuDraw();
        break;
}

// Draw the current display message
if (gameMessages.Count > 0)
{
    DisplayMessage currentMessage = gameMessages[0];
    spriteBatch.Begin();
    Vector2 stringSize = spriteFont.MeasureString(gameMessages[0].Message);
    spriteBatch.DrawString(spriteFont, gameMessages[0].Message, new Vector2((1280 -
stringSize.X) / 2.0f, 500), Color.White);
    spriteBatch.End();
}
```

For now, you just call the `MainMenuDraw` to draw the main menu. Later in the chapter, you add draw methods for all of the value of `GameState`. The `Draw` method also displays the current `DisplayMessage` at the bottom of the screen.

To implement `MainMenuDraw` to display the text for the main menu, add the following method to your `Game` class.

```
// Draw method for the MainMenu GameState
private void MainMenuDraw()
{
    spriteBatch.Begin();
    spriteBatch.DrawString(spriteFont, "MAIN MENU", new Vector2(10, 10),
➥Color.White);
    spriteBatch.DrawString(spriteFont, "Create Session - Press A",
                           new Vector2(10, 50), Color.White);
    spriteBatch.DrawString(spriteFont, "Find Session - Press B",
                           new Vector2(10, 90), Color.White);
    spriteBatch.DrawString(spriteFont, "Exit - Press Back",
                           new Vector2(10, 130), Color.White);
    spriteBatch.End();
}
```

The main menu text is simple and displays the menu title and three menu options for creating a session, finding a session, and exiting the game.

Running the sample at this point should just display the simple main menu. The `A` and `B` buttons won't do anything yet because you have not wired them in your `MainMenuUpdate` method. Pressing the Back button exits the sample.

The sample should look like Figure 16.4.

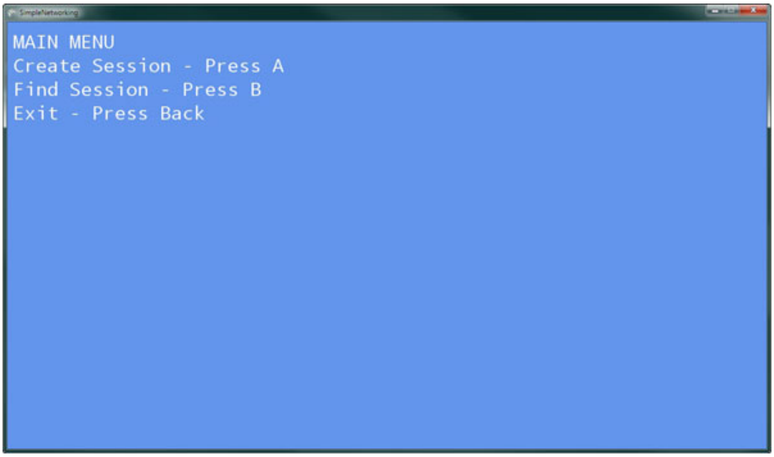
Creating a Network Session

Although your main menu has text that says, "press `A` to create a session," pressing the button does not do anything. To change this, add the following lines of code to the `MainMenuUpdate` method:

```
// Create a new session
if (ButtonPressed(Buttons.A))
    gameState = GameState.CreateSession;
```

This causes oyur game to change to the `CreateSession` state after the user presses the `A` button.

Because the game is in `CreateSession` state, add to the switch statements in the `Update` and `Draw` methods so they know what to do while you are in this state.



MAIN MENU
Create Session - Press A
Find Session - Press B
Exit - Press Back

Figure 16.4 Main menu screen

In the game's `Draw` method, add the following to the switch statement:

```
case GameState.CreateSession:
    CreateSessionDraw();
    break;
```

The `CreateSessionDraw` handles drawing the menu screen when the game is in the `CreateSession` state. Add the `CreateSessionDraw` method to your game.

```
// Draw method for the CreateSession GameState
private void CreateSessionDraw()
{
    spriteBatch.Begin();
    spriteBatch.DrawString(spriteFont, "CREATE SESSION",
                           new Vector2(10, 10), Color.White);
    spriteBatch.DrawString(spriteFont, "Deathmatch - Press A",
                           new Vector2(10, 50), Color.White);
    spriteBatch.DrawString(spriteFont, "CaptureTheFlag - Press B",
                           new Vector2(10, 90), Color.White);
    spriteBatch.DrawString(spriteFont, "FreeForAll - Press X",
                           new Vector2(10, 130), Color.White);
    spriteBatch.DrawString(spriteFont, "Exit - Press Back",
                           new Vector2(10, 170), Color.White);
    spriteBatch.End();
}
```

In the game's `Update` method, add the following lines of code to the switch statement:

```
case GameState.CreateSession:
    CreateSessionUpdate();
    break;
```

While the game is in the `CreateSession` state, call the `CreateSessionUpdate` method to run the update logic for the state. Add the following method to your `Game` class:

```
// Update method for the create session method
private void CreateSessionUpdate()
{
    // Move back to the main menu
    if (ButtonPressed(Buttons.Back))
        gameState = GameState.MainMenu;
    // Create a new session with different
    // using different values for GameType
    else if (ButtonPressed(Buttons.A))
        CreateSession(GameType.Deathmatch);
    else if (ButtonPressed(Buttons.B))
        CreateSession(GameType.CaptureTheFlag);
    else if (ButtonPressed(Buttons.X))
        CreateSession(GameType.FreeForAll);
}
```

`CreateSessionUpdate` checks for button presses to create different game types by calling the `CreateSession` method, which you create in a moment. The `GameType` is an enumeration that you create for the game. Add the following enumeration to your game:

```
public enum GameType { Deathmatch, CaptureTheFlag, FreeForAll };
```

Before you implement `CreateSession`, you need an additional enumeration and some fields added to your game. Add the following enumeration to your game:

```
public enum SessionProperties { GameType, MapLevel, OtherCustomProperty };
```

When you create a session, you can provide a number of integer values that will define what type of session to create. A helpful way to keep track of integer values is to use an enumeration. In this case, create `SessionProperties`, which defines three types of properties.

You also need to add a member variable to store the `NetworkSession`. The `NetworkSession` object is what you use to manage the active session and to send and receive data over the session. You can think of a session as the connection of your machine with the other machines. There are times that gamers join and times where they leave, but there is always a host to the session that controls the current state of the session. If the host leaves the session, the session is over unless host migration is turned on. In that case, a new host is selected. Add the following member variable to your game:

```
// The network session for the game
NetworkSession networkSession;
```

Now you can create the `CreateSession` method that creates a new session given a `GameType`. The `GameType` is passed in depending on the selection the player has made from the create session menu screen.

```
// Createa a new NetworkSession
private void CreateSession(GameType gameType)
{
    // If we have an existing network session we need to dispose of it
    if (networkSession != null && !networkSession.IsDisposed)
        networkSession.Dispose();

    // Create the NetworkSessionProperties to use for the session
    // Other players will use these to search for a session
    NetworkSessionProperties sessionProperties = new NetworkSessionProperties();
    sessionProperties[(int)SessionProperties.GameType] = (int)gameType;
    sessionProperties[(int)SessionProperties.MapLevel] = 0;
    sessionProperties[(int)SessionProperties.OtherCustomProperty] = 42;

    // Create the NetworkSession NetworkSessionType of SystemLink
    networkSession = NetworkSession.Create(NetworkSessionType.SystemLink, 1, 4, 0,
                                       sessionProperties);
    networkSession.AllowJoinInProgress = true;

    // Register for NetworkSession events
    networkSession.GameStarted +=
            new EventHandler<GameStartedEventArgs>(networkSession_GameStarted);
    networkSession.GameEnded +=
            new EventHandler<GameEndedEventArgs>(networkSession_GameEnded);
    networkSession.GamerJoined +=
            new EventHandler<GamerJoinedEventArgs>(networkSession_GamerJoined);
    networkSession.GamerLeft +=
            new EventHandler<GamerLeftEventArgs>(networkSession_GamerLeft);
    networkSession.SessionEnded +=
            new
EventHandler<NetworkSessionEndedEventArgs>(networkSession_SessionEnded);

    // Move the game into the GameLobby state
    gameState = GameState.GameLobby;
}
```

Before you create the session, check whether you need to dispose of an already existing session.

When you create a session, you provide a set of `NetworkSessionProperties`. These properties define how the session is used. For example, your game might have multiple types of multiplayer games, different levels, or other values that define what is happening in the session. These values are important because they are used when you search for available sessions. The values can also be displayed in the search results when players are looking for a session to join.

The different values of the `SessionProperties` enumeration are used to index into the `NetworkSessionProperties`. We give some simple values for the `MapLevel` and

`OtherCustomProperty` just to show you how this is done. If this was a real game, you would use the player's map selection and other logic to drive these values.

Now you are ready to create the `NetworkSession`. The `Create` method takes a number of parameters and has a few overloads. The first value is the `NetworkSessionType` to create. As the name of the enumeration implies, this is the type of network session to create. There are four main types of sessions.

The first type of `NetworkSessionType` is `Local`. As the name implies, the `NetworkSession` is local only to a single machine. This is useful if you create a networked game but then want to use the same networking code for your split screen game.

`SystemLink` is the type of `NetworkSessionType` that you are using in this sample and is most likely what you will use while in development. It enables multiple machines that are on the same local area network to connect to each other.

`PlayerMatch` and `Ranked` are Xbox LIVE sessions that enable gamers to play across the world together.

The second parameter to the `NetworkSession.Create` method is the number of local gamers that participate in the session. Your game can allow for multiple players on the same console to join other players in a networked game. When you join a session, you and any other local gamers take up multiple spots in the session. In the sample, we set this value to 1 for just a single local user.

The third parameter is the max number of gamers for the entire session. XNA games can support up to 31 maximum players per session. Although 31 players are possible, determine what makes sense for your game and what type of performance you get with multiple players in the session.

The fourth parameter is the amount of private slots. A private slot is one that can be filled only by an invite from a player in the session. This is helpful for times where players are trying to have a game with their friends and don't want other players to join.

The final parameter is the `NetworkSessionProperties` that you created earlier. The `Create` method returns to you a new `NetworkSession`.

> **Note**
>
> `NetworkSession.Create` can block for a number of frames. XNA provides asynchronous `BeginCreate` and `EndCreate` methods to prevent your game from skipping frames.

After creating the `NetworkSession`, set the `AllowJoinInProgress` to `true`. This property enables players joining the session to do so even if the game has already started. Notice later in the chapter that joining a session can take place in the lobby or while playing the game. If you don't want to allow players to join the game while it is ongoing, you can set `AllowJoinInProgress` to `false`, which keeps them from connecting to the session.

Now that the new `NetworkSession` is created, register for a number of helpful events that will occur during the lifespan of the `NetworkSession`.

The `GameStarted` and `GameEnded` events fire when the session starts and ends a game. A game is different than a session. A session is the connection of the players while a game represents a play session. For examples, you and your friends can stay in one session while you play multiple games that start and finish over and over again. These events are raised when the `NetworkSession.StartGame` and `NetworkSession.EndGame` methods are called. Add the following two event handlers to your game:

```
// Event handler for the NetworkSession.GameStarted event
// This event is fired when the host call NetworkSession.StartGame
void networkSession_GameStarted(object sender, GameStartedEventArgs e)
{
    gameMessages.Add(new DisplayMessage("Game Started", TimeSpan.FromSeconds(2)));
    // Move the game into the PlayingGame state
    gameState = GameState.PlayingGame;
}

// Event handler for the NetworkSession.GameEnded event
// This event is fired when the host call NetworkSession.EndGame
void networkSession_GameEnded(object sender, GameEndedEventArgs e)
{
    gameMessages.Add(new DisplayMessage("Game Ended", TimeSpan.FromSeconds(2)));
    // Move the game into the GameLobby state
    gameState = GameState.GameLobby;
}
```

When the game starts or ends, print a message and set the next `GameState`.

The `GamerJoined` and `GamerLeft` events fire when a gamer joins or leaves the session. Even the host gets the `GamerJoined` event. Add the following two event handlers to your game:

```
// Event handler for the NetworkSession.GamerJoined event
// This event is fired when someone joins the session
// This event will fire even for local gamers
void networkSession_GamerJoined(object sender, GamerJoinedEventArgs e)
{
    gameMessages.Add(new DisplayMessage("Gamer Joined: " + e.Gamer.Gamertag,
                                        TimeSpan.FromSeconds(2)));

    // Add a new GameObject that we will use to store game state for the player
    e.Gamer.Tag = new GameObject(new Vector2(random.Next(100, 1000),
                                             random.Next(100, 600)));
}

// Event handler for the NetworkSession.GamerLeft event
// This event is fired when a player leaves the session
void networkSession_GamerLeft(object sender, GamerLeftEventArgs e)
{
    gameMessages.Add(new DisplayMessage("Gamer Left: " + e.Gamer.Gamertag,
                                        TimeSpan.FromSeconds(2)));
}
```

For both the `GamerJoined` and `GamerLeft` events, you print a message with the `Gamertag`. The `GamerJoined` handler also adds a new `GameObject` to the `Tag` property of the `Gamer`. The Tag property enables you to store any type of `object` on an instance of a `Gamer`. You use a simple class called `GameObject`, which stores the current position of the player. In your game, you can store all of the players' current state in this type.

Define the `GameObject` class in your game using the following code:

```
// Each gamer in the session will have their own GameObject
public class GameObject
{
    public Vector2 Position
    { get; set; }

    public GameObject(Vector2 position)
    {
        Position = position;
    }
}
```

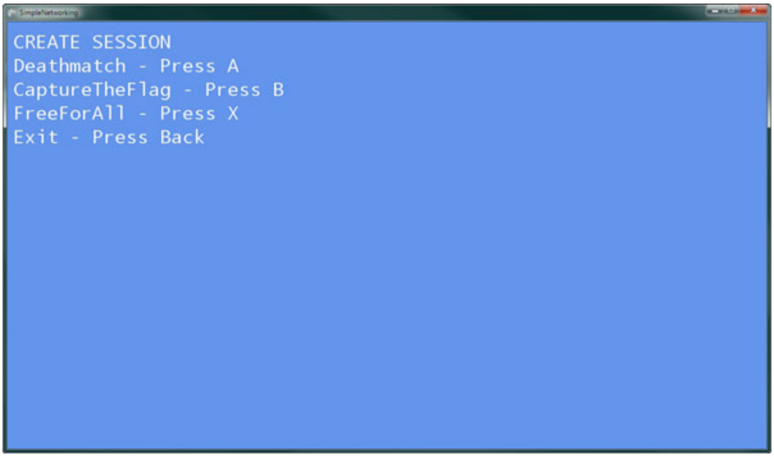
The final event `SessionEnded` is fired when a player leaves the session. Add the following event handler to your game:

```
// Event handler for the NetworkSession.SessionEnded event
// This event is fired when your connection to the NetworkSession is ended
void networkSession_SessionEnded(object sender, NetworkSessionEndedEventArgs e)
{
    gameMessages.Add(new DisplayMessage("Session Ended: " + e.EndReason.ToString(),
                                        TimeSpan.FromSeconds(2)));

    // Since we have disconnected we clean up the NetworkSession
    if (networkSession != null && !networkSession.IsDisposed)
        networkSession.Dispose();

    // Move the game into the MainMenu state
    gameState = GameState.MainMenu;
}
```

After the session ends, dispose of the `NetworkSession` and set the `gameState` to go back to the `MainMenu`.

After the `CreateSession` has completed, you are in the `GameLobby` state waiting for others to join and for the game to start.

Running the game now should enable you to go to the create session screen and see something similar to Figure 16.5.



CREATE SESSION
Deathmatch - Press A
CaptureTheFlag - Press B
FreeForAll - Press X
Exit - Press Back

Figure 16.5 Create session menu screen

The `NetworkSession` needs to be updated to allow for data to be sent and received to all of the players in the session. To do this, call `NetworkSession.Update`. In the game's `Update` method, add the following code:

```
// Update the network session we need to check to
// see if it is disposed since calling Update on
// a disposed NetworkSession will throw an exception
if (networkSession != null && !networkSession.IsDisposed)
    networkSession.Update();
```

If the session is not null and has not been disposed, call the `Update` method.

Building a Game Lobby

With the session created, you move into the game lobby that lists the players in the session and whether they are ready to play the game until the host starts the game.

The game is in `GameLobby` state, so add to the switch statements in the `Update` and `Draw` methods so they know what to do while you are in this state.

In the game's `Draw` method, add the following to the switch statement.

```
case GameState.GameLobby:
    GameLobbyDraw();
    break;
```

The `GameLobbyDraw` method draws the current players in the session and their current ready and talking state. Add the following method to your game:

```
// Draw method for the GameLobby GameState
private void GameLobbyDraw()
{
    spriteBatch.Begin();
    spriteBatch.DrawString(spriteFont, "GAME LOBBY", new Vector2(10, 10),
➥Color.White);
    spriteBatch.DrawString(spriteFont, "Mark Ready Status - Press A",
                                       new Vector2(10, 50), Color.White);
    spriteBatch.DrawString(spriteFont, "Exit - Press Back",
                                       new Vector2(10, 90), Color.White);
    if (networkSession.IsHost)
        spriteBatch.DrawString(spriteFont, "Start Game - Press Start",
                                           new Vector2(10, 130), Color.White);

    // Draw all games in the lobby
    spriteBatch.DrawString(spriteFont, "PLAYERS IN LOBBY",
                                       new Vector2(10, 220), Color.White);
    float drawOffset = 0;
    foreach (NetworkGamer networkGamer in networkSession.AllGamers)
    {
        spriteBatch.DrawString(spriteFont, networkGamer.Gamertag + " - " +
                                            ((networkGamer.IsReady) ? "READY" :
                                                                      "NOT
➥READY") + " " +
                                             ((networkGamer.IsTalking) ?
➥"TALKING" : ""),
                                            new Vector2(10, 260 + drawOffset),
                                            Color.White);

        drawOffset += 40;
    }

    spriteBatch.End();
}
```

`GameLobbyDraw` first draws the menu title of GAME LOBBY and instructions on how to mark your ready status by pressing the `A` button and how to exit by pressing the `Back` button. If the machine is the host, then instructions on how to start the game are also displayed.

It then loops over all of the `NetworkGamer` instances in the session by using the `AllGamers` property of the `NetworkSession`. You then write out their `Gamertag` and whether they are ready and whether they are talking. Every gamer can mark whether he or she is ready to play the game or not. This is just a status indication and does not prevent the host from starting the game.

In the game's `Update` method, add the following to the switch statement.

```
case GameState.GameLobby:
    GameLobbyUpdate();
    break;
```

The `GameLobbyUpdate` method enables players to exit the lobby and mark whether they are ready to play or not. If the machine is also the host, then you also allow for the host to start the game. Add the following method to your game:

```
// Update method for the GameLobby GameState
private void GameLobbyUpdate()
{
    // Move back to the main menu
    if (ButtonPressed(Buttons.Back))
    {
        networkSession.Dispose();
        gameState = GameState.MainMenu;
    }
    // Set the ready state for the player
    else if (ButtonPressed(Buttons.A))
        networkSession.LocalGamers[0].IsReady = !networkSession.LocalGamers[0].
➥IsReady;
    // Only the host can start the game
    else if (ButtonPressed(Buttons.Start) && networkSession.IsHost)
        networkSession.StartGame();
}
```

`NetworkSession` provides a property called `LocalGamers`, which contains `SignedInGamer` references for all of the players in the session that are local to the machine. Because you know that there is only one local player in the session, you can use the 0 index. If you allow multiple players, check which controller pressed a button and update the correct index. Setting the `IsReady` property sends the state to all of the players in the session.

If the host presses the Start button, then the game starts by calling the `NetworkSession.StartGame` method. When the game begins, the `GameStarted` event fires and the event handler updates the current `GameState` to `PlayingGame`.

Running the game now enables you to create the session and join the lobby, which should look similar to Figure 16.6.

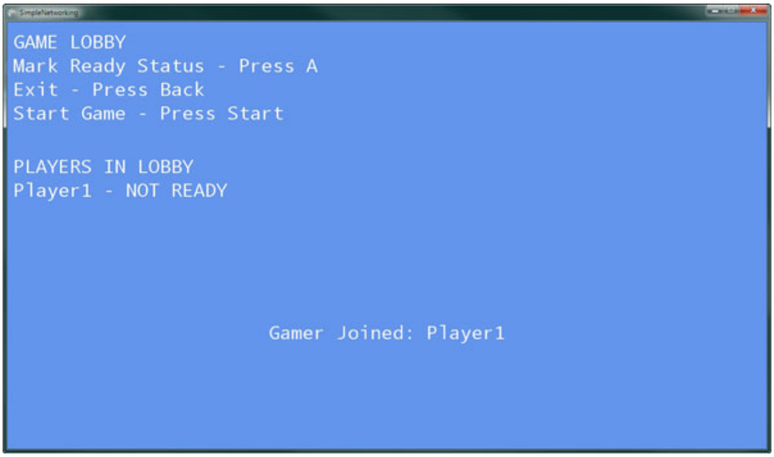
Playing the Game

Now that the game has started, take the local users' input and update the position of their `GameObject` to move across the screen. Then, send the local players' state to all of the players in the session and read any state that has been sent to you from other players. The game is quite simple—draw the players' `Gamertag` at their position.

Because the game is in `PlayingGame` state, add to the switch statements in the `Update` and `Draw` methods so they know what to do while you are in this state.

In the game's `Draw` method, add the following to the switch statement:

```
case GameState.PlayingGame:
    PlayingGameDraw();
    break;
```



GAME LOBBY
Mark Ready Status - Press A
Exit - Press Back
Start Game - Press Start

PLAYERS IN LOBBY
Player1 - NOT READY

Gamer Joined: Player1

Figure 16.6 Game lobby menu screen

Now add the `PlayingGameDraw` method to your game using the following code:

```
// Draw method for the PlayingGame GameState
private void PlayingGameDraw()
{
    spriteBatch.Begin();
    spriteBatch.DrawString(spriteFont, "PLAYING GAME", new Vector2(10, 10),
➥Color.White);

    // Draw each players name at their position in the game
    foreach (NetworkGamer networkGamer in networkSession.AllGamers)
    {
        GameObject player = networkGamer.Tag as GameObject;

        spriteBatch.DrawString(spriteFont, networkGamer.Gamertag +
                                          ((networkGamer.IsHost) ? " (HOST)": "") +
                                             ((networkGamer.IsTalking) ? "
➥TALKING" : ""),
                                            player.Position,
                                            Color.White);
    }

    spriteBatch.End();
}
```

The `PlayingGameDraw` method loops through all of the `NetworkGamer` instances in the `NetworkSession.AllGamers` collection, which contains all of the players in the session. The `Gamertag` of each gamer is draw to the screen at the position stored in the

`GameObject` for the gamer. If the gamer is the host or is talking, display text next to the `Gamertag` that shows those values.

To send and receive data, you need two additional member variables for your game. Add the following member variables to your game:

```
// PacketWriter and PacketReader used to send and recieve game data
PacketWriter packetWriter = new PacketWriter();
PacketReader packetReader = new PacketReader();
```

The `PacketWriter` and `PacketReader` are helper classes that help with sending and receiving data across the `NetworkSession`. Use these to write and read each player's state.

In the game's `Update` method, add the following to the switch statement.

```
case GameState.PlayingGame:
    PlayingGameUpdate(gameTime);
    break;
```

The `PlayingGameUpdate` method is responsible for updating the game state for each of the local players and sending their state to all of the players in the session. The state of the other players in the session is read and their `GamerObject` is updated to their current state. Add the following method to your game:

```
// Update method for the PlayingGame GameState
private void PlayingGameUpdate(GameTime gameTime)
{
    // Check to see if the player wants to quit
    if (ButtonPressed(Buttons.Back))
    {
        // If the player is the host then
        // the game is exited but the session
        // stays alive
        if (networkSession.IsHost)
            networkSession.EndGame();
        // Other players leave the session
        else
        {
            networkSession.Dispose();
            networkSession = null;
            gameState = GameState.MainMenu;
        }

        return;
    }

    // Use GamePad to update the position of the local gamers
    // Loop all of the local gamers
    foreach (LocalNetworkGamer gamer in networkSession.LocalGamers)
    {
        // Handle local input
        GamePadState gamePadState = GamePad.GetState(gamer.SignedInGamer.
➥PlayerIndex);

        // Get the GameObject for this local gamer
        GameObject gameObject = gamer.Tag as GameObject;
        Vector2 position = gameObject.Position;

        // Update the position of the game object based on the GamePad
        Vector2 move = gamePadState.ThumbSticks.Left *
                       (float)gameTime.ElapsedGameTime.TotalSeconds * 500;
        move.Y *= -1;
        position += move;
        gameObject.Position = position;

        // Write the new position to the packet writer
        packetWriter.Write(position);

        // Send the data to everyone in your session
        gamer.SendData(packetWriter, SendDataOptions.InOrder);
    }

    // Read data that is sent to local players
    foreach (LocalNetworkGamer gamer in networkSession.LocalGamers)
    {
        // Read until there is no data left
        while (gamer.IsDataAvailable)
        {
            NetworkGamer sender;
            gamer.ReceiveData(packetReader, out sender);

            // We only need to update the state of non local players
            if (sender.IsLocal)
                continue;

            // Get GameObject of the sender
            GameObject gameObject = sender.Tag as GameObject;

            // Read the position
            gameObject.Position = packetReader.ReadVector2();
        }
    }
}
```

First, `PlayingGameUpdate` checks whether the player has pressed the `Back` button asking to quit the game. If the current machine is the host, then `NetworkSession.EndGame` is called, which fires the corresponding event that sends everyone in the game back to the lobby. If the player is not the host, then dispose of the `NetworkSession` and set the state to return back to the `MainMenu`.

Next, all of the `LocalGamers` are looped and the `GameObject.Position` value is updated using the `GamePad` input. After the position of the local gamer is updated, the new position is written to the `PacketWriter` and set to everyone in the session using `NetworkSession.SendData`.

There are a number of overloads to `SendData`—the one in this sample takes two parameters. The first is the data to send which takes the `PacketWriter`. The second defines how the data should be sent using the `SendDataOptions` enumeration.

The Internet is a like a major highway with a lot of traffic. That traffic can be moving fast or can get into traffic jams. There are also multiple routes from one point to another. When you send data across the network, the data is broken down into packets. Each packet is sent from your machine to another across the network. Depending on the network conditions, some packets can get lost and never get to the recipient; others might get stalled and get behind packets sent after themselves.

`SendDataOptions` enables you to specify how you would like the data to be handled. XNA can guarantee that packets be delivered by checking whether the other machine receives the packet and sending the packet again until it is received. As you can imagine, this creates more overhead because there is more data sent that ensures the packets are received. If you want to guarantee that the packets are received, use the `Reliable` option. XNA can also guarantee that the packets are received in order. If you want the packets to arrive in order, then use the `InOrder` option. You can also specify `ReliableInOrder`, which guarantees the data is both in order and is delivered. If you aren't concerned about lost packets and data order, use the `SendDataOptions.None` option.

> **Note**
>
> An overload of `NetworkSession.SendData` enables you to specify a specific `NetworkGamer` to send the packets to.

Now that the data is sent, read the data that was sent to your local gamers. Remember that the `LocalGamers` are looped. The `LocalNetworkGamer.IsDataAvailable` property is used to check whether there is more data to read for a specific `LocalNetworkGamer`. `LocalNetworkGamer.ReceiveData` is called to read the data sent from other players. `ReceiveData` takes two parameters. The first is the `PacketReader` and the second is the `NetworkGamer` who sent the data.

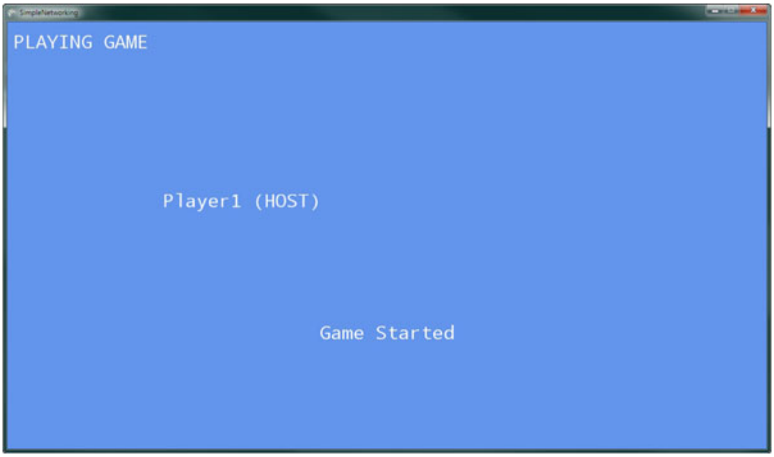
The sender is checked to see whether it is local to the session using the `IsLocal` property. If it is a local gamer, then you don't need to update the position because you would have already done so previously. If the data came from another player in the session and was not local, then the `GameObject` for that player is updated with the position read from the `PacketReader`.

Note

There are multiple ways you can set up your game to send data across the network. This sample uses a technique called peer to peer, which sends all of the data from each user in the session to all other players in the session.

Another way you can send the data is called client-server model. With client-server, there is one machine that contains the current state of the game and sends that state to all of the other machines in the network called clients. Each client talks only to one machine, which is the server.

Now that you are sending and receiving data, the game is ready to be played. Running the sample now should enable you to create a session and start the game from the lobby. Figure 16.7 shows the host playing the game alone.



PLAYING GAME

Player1 (HOST)

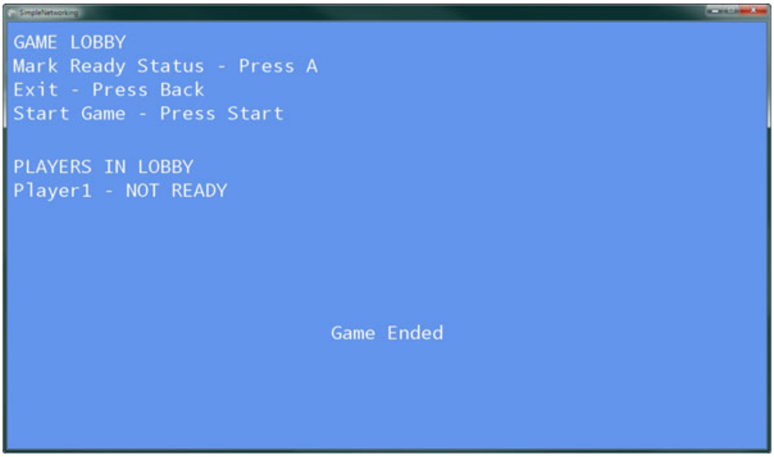
Game Started

Figure 16.7 Host playing the game

If you press the Back button, you quit the game and are returned back into the lobby as shown by Figure 16.8.

Searching for an Available Network Session

So now you can create a `NetworkSession` and start the game, but you are missing the capability for other players to join your game. You need to search for the available network sessions and then join one of the sessions. After you join, you have already created the code to handle playing and quitting the game.



GAME LOBBY
Mark Ready Status - Press A
Exit - Press Back
Start Game - Press Start

PLAYERS IN LOBBY
Player1 - NOT READY

Game Ended

Figure 16.8 Quitting a game back to the lobby

To wire up the find session logic, add code to the `MainMenuUpdate` method to check for the `B` button that is pressed.

```
// Find a session
if (ButtonPressed(Buttons.B))
    FindSession();
```

When the player presses the `B` button at the main menu, start the search for a session to join.

To store the collection of `AvailableNetworkSession` objects so that you can pick the one you want to join, add the following member variable to your `Game` class:

```
// List of sessions that you can join
AvailableNetworkSessionCollection availableSessions;
```

Now you need to implement the `FindSession` method, which populates the `AvailableNetworkSessionCollection` and switches out `GameState` to the `FindSession` menu screen. Add the following method to your `Game` class:

```
// Method to start the search for a NetworkSession
private void FindSession()
{
    // Dispose of any previous session
    if (networkSession != null && !networkSession.IsDisposed)
        networkSession.Dispose();

    // Define the type of session we want to search for using the
    // NetworkSessionProperties.
    // We only set the MapLevel and the OtherCustomProperty
    NetworkSessionProperties sessionProperties = new NetworkSessionProperties();
    sessionProperties[(int)SessionProperties.MapLevel] = 0;
    sessionProperties[(int)SessionProperties.OtherCustomProperty] = 42;

    // Find an available NetworkSession
    availableSessions = NetworkSession.Find(NetworkSessionType.SystemLink,
                                            1, sessionProperties);

    // Move the game into the FindSession state
    gameState = GameState.FindSession;
}
```

`FindSession` first checks whether there is a current active `NetworkSession` and calls Dispose on the session if it is active.

As you did when you created the `NetworkSession`, create a `NetworkSessionProperties` object. This time, the properties are used to search for a specific session. You don't need to specify all of the properties. For example, you don't set a `GameType` value so the sessions returned can be of any `GameType`.

To generate the list of `AvailableNetworkSession` instances, call the `NetworkSession.Find` method. `Find` takes three arguments. The first is the `NetworkSessionType` just like you used for the `Create` method. The second is the number of local players that will be joining you in the session. Specify just 1 for this example. Finally the third and final argument is the `NetworkSessionProperties` to use in the search.

The resulting `AvailableNetworkSessionCollection` contains all of the `AvailableNetworkSession` instances that match the properties and have room for you to join.

> **Note**
>
> Like the `NetworkSession.Create` method, the `NetworkSession.Find` also has asynchronous versions called `BeginFind` and `EndFind`.

The last thing `FindSession` does is set the `gameState` to `GameState.FindSession` to display the find session menu screen.

Add to the switch statements in the `Update` and `Draw` methods so gamers know what to do while you are in this state.

In the game's `Draw` method, add the following to the switch statement:

```
case GameState.FindSession:
    FindSessionDraw();
    break;
```

Now, add the `FindSessionDraw` method to your game using the following code:

```
// Draw method for the FindSession GameState
private void FindSessionDraw()
```

```
{
    spriteBatch.Begin();
    spriteBatch.DrawString(spriteFont, "FIND SESSION", new Vector2(10, 10),
➥Color.White);
    spriteBatch.DrawString(spriteFont, "Exit - Press Back",
                           new Vector2(10, 50), Color.White);

    // Write message if there are no sessions found
    if (availableSessions.Count == 0)
    {
        spriteBatch.DrawString(spriteFont, "NO SESSIONS FOUND",
                               new Vector2(10, 90), Color.White);
    }
    else
    {
        // Print out a list of the available sessions
        int sessionIndex = 0;
        foreach (AvailableNetworkSession session in availableSessions)
        {
            spriteBatch.DrawString(spriteFont, session.HostGamertag + " " +
                                   session.OpenPublicGamerSlots +
                                   ((sessionIndex == 0) ? " (PRESS A)" : ""),
                                   new Vector2(10, 90 + sessionIndex * 40),
➥Color.White);
            sessionIndex++;
        }
    }

    spriteBatch.End();
}
```

The `FindSessionDraw` method loops over all of the `AvailableNetworkSession` instances in the `AvailableNetworkSessionCollection` and writes out the host `Gamertag` and how many gamer slots are available.

`AvailableNetworkSession` provides many properties that expose information about the `NetworkSession` that you can join. `HostGamertag` returns the string `Gamertag` for the host of the session. This might not be the person who started the session because the session might support host migration. `CurrentGamerCount`, `OpenPrivateGamerSlots`, and `OpenPublicGamerSlots` return the total gamers in the session, and the amount of open private and public slots to join. The `SessionProperties` property returns the `NetworkSessionProperties` used to create the session. Finally, the `QualityOfService` property returns an instance of `QualityOfService`, which tells you how well you can connect to the `AvailableNetworkSession`.

The `QualityOfService` class provides helpful measurements that enable you to see how well the connection of the machine is to the `AvailableNetworkSession`. `QualityOfService` provides a number of properties including `AverageRoundtripTime` and `MinimumRoundtripTime`, which returns the average and minimum amount of time it takes for the machine to take to the host of the `NetworkSession`. The `BytesPerSecondDownstream` and `BytesPerSecondUpstream` properties are the estimated download and upload bandwidth from the machine making the call to the host of the `NetworkSession`.

Next, update the game's `Update` method with another condition in the switch statement to handle updating while you are in the `FindSession` state. Add the following to the `switch` statement in the Update method:

```
case GameState.FindSession:
    FindSessionUpdate();
    break;
```

Now, add the `FindSessionUpdate` method to the `Game` class:

```
// Update method for the FindSession GameState
private void FindSessionUpdate()
{
    // Go back to the main menu
    if (ButtonPressed(Buttons.Back))
        gameState = GameState.MainMenu;
    // If the user presses the A button join the first session
    else if (ButtonPressed(Buttons.A) && availableSessions.Count != 0)
        JoinSession(0);
}
```

To keep the sample simple, you don't handle selecting all of the available network sessions and just pick the first one. In your game, take user input to move some type of selection to enable the user to select which session to join.

Now you come to the point where you need to run the sample on multiple machines. You need one machine to create the session and another to search for available sessions.

Open another instance of Visual Studio and open the same solution file. If you have not already created an Xbox 360 version of the project, you should create a copy of the project for Xbox 360.

In one of the instances of Visual Studio, launch the Windows version of the sample project. Have the player create a session and wait at the lobby. In the other instance of Visual Studio, launch the Xbox 360 version of the sample project. Select the find session menu option. Now you should see something similar to Figure 16.9 showing the available sessions.

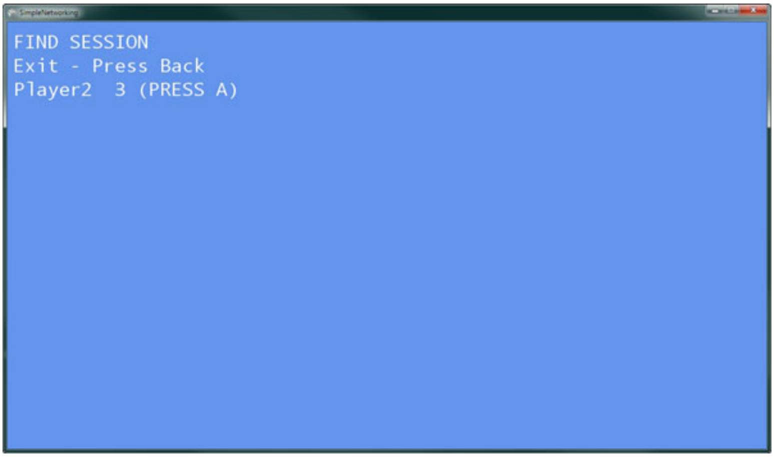

FIND SESSION
Exit - Press Back
Player2 3 (PRESS A)

Figure 16.9 Find session menu screen

Joining an Available Network Session

Finally, implement the `JoinSession` method, which joins one of the available network sessions, registesr for session events, and sets the next `GameState`. Add the following method to your game:

```
// Creates a session using a AvailableNetworkSession index
private void JoinSession(int sessionID)
{
    // Join an existing NetworkSession
    try
    {
        networkSession = NetworkSession.Join(availableSessions[sessionID]);
    }
    catch (NetworkSessionJoinException ex)
    {
        gameMessages.Add(new DisplayMessage("Faild to connect to session: " +
                                            ex.JoinError.ToString(),
                                            TimeSpan.FromSeconds(2)));

        // Check for sessons again
        FindSession();
    }
    // Register for NetworkSession events
    networkSession.GameStarted +=
             new EventHandler<GameStartedEventArgs>(networkSession_GameStarted);
    networkSession.GameEnded +=
             new EventHandler<GameEndedEventArgs>(networkSession_GameEnded);
    networkSession.GamerJoined +=
              new EventHandler<GamerJoinedEventArgs>(networkSession_GamerJoined);
    networkSession.GamerLeft +=
              new EventHandler<GamerLeftEventArgs>(networkSession_GamerLeft);
    networkSession.SessionEnded +=
              new EventHandler<NetworkSessionEndedEventArgs>(networkSession
➥SessionEnded);

    // Set the correct GameState. The NetworkSession may have already started a
game.
    if (networkSession.SessionState == NetworkSessionState.Playing)
        gameState = GameState.PlayingGame;
    else
        gameState = GameState.GameLobby;
}
```

The `JoinSession` method that you added to your game takes the index value into the `availableSessions` collection to use when joining the session. The `NetworkSession.Join` method takes a single argument, which is the `AvailableNetworkSession` to join. Join can throw a `NetworkSessionJoinException` if for some reason the session is not joinable anymore—for example, if the session becomes full between searching for available sessions and trying to join the session. If you receive the exception, you search again and print an error message.

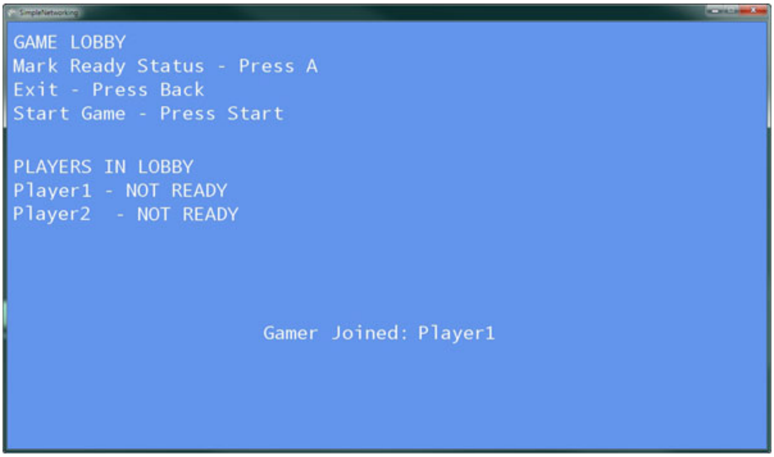
After joining the session, you have a valid `NetworkSession` similar to when you created the session, so register for all of the same events. Finally, set the current `gameState` depending on whether the session you are joining is already playing the game or is in the lobby.

> **Note**
>
> As with the `NetworkSession.Create` and `NetworkSession.Find` methods, `NetworkSession.Join` also has an asynchronous version `BeginJoin` and `EndJoin`.

Running the sample now should enable you to create a session on one machine and join that session from another. The lobby should look similar to Figure 16.10.

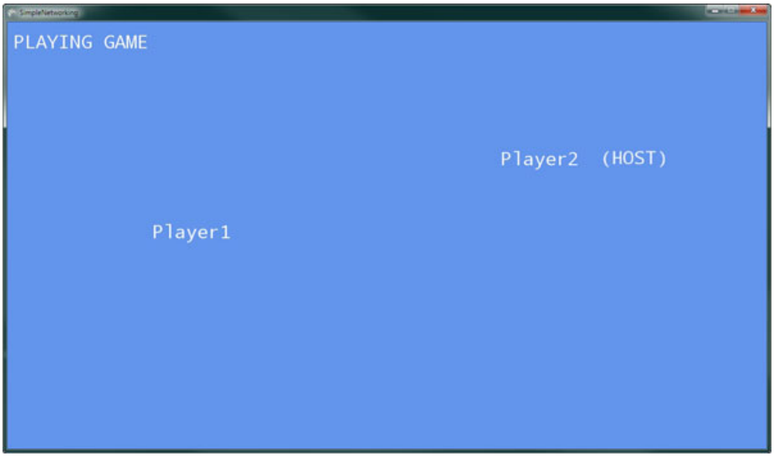
Both players can set their ready state, which should update on both machines. If the host presses the `Start` button, the game begins. In the game playing state, both players can move their name around the screen (see Figure 16.11).

If the host presses the `Back` button, the game ends and all players move back into the lobby. If the host presses the `Back` button again, the session ends and all players return to the main menu (see Figure 16.12).


SimpleNetworking

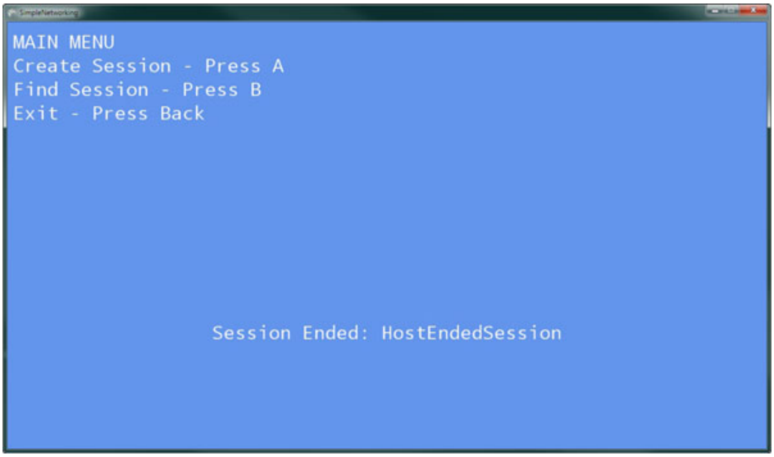
```
GAME LOBBY
Mark Ready Status - Press A
Exit - Press Back
Start Game - Press Start

PLAYERS IN LOBBY
Player1 - NOT READY
Player2  - NOT READY



                    Gamer Joined: Player1
```

Figure 16.10 Game lobby with multiple players


SimpleNetworking

```
PLAYING GAME


                                        Player2  (HOST)

           Player1
```

Figure 16.11 Two players moving their names around the screen



MAIN MENU
Create Session - Press A
Find Session - Press B
Exit - Press Back

Session Ended: HostEndedSession

Figure 16.12 Host exiting the session sends all players to the main menu

Sending Player Invites

Xbox LIVE enables you to connect with players from around the world. While playing a game, you might want to invite your friends to join you. You might also receive invitations from others to join their game. Adding support for game invites is tremendously easy with XNA Game Studio.

In the `GameLobbyUpdate` method, add the following code:

```
// Invite other players
if (ButtonPressed(Buttons.X) && !Guide.IsVisible)
    Guide.ShowGameInvite(PlayerIndex.One, null);
```

`Guide.ShowGameInvite` shows a list of the player's friends to invite to the current game.

Now, you need to register for the `NetworkSession.InviteAccepted` event, which fires after a player accepts the invite.

```
// Register for the invite event
NetworkSession.InviteAccepted +=
                new EventHandler<InviteAcceptedEventArgs>(NetworkSession
➥InviteAccepted);
```

Finally, implement the `NetworkSession_InviteAccepted` method.

```
void NetworkSession_InviteAccepted(object sender, InviteAcceptedEventArgs e)
{
    if (networkSession != null && !networkSession.IsDisposed)
        networkSession.Dispose();

    if (!e.IsCurrentSession)
        networkSession = NetworkSession.JoinInvited(1);
}
```

First, `NetworkSession_InviteAccepted` disposes of any `NetworkSession` that might be active. Then, check to make sure you are not already in the session before you call the `NetworkSession.JoinInvited` method. This method takes the number of local gamers that are joining. Now, your game is ready for game invites.

> **Note**
>
> If the account you are using is not an Xbox LIVE account, then the `Guide.ShowGameInvite` method throws an exception because local players can't send invites.

Party Invites

A party allows for up to eight players to group together no matter which game they are playing. The players in the party can chat with each other even while playing different games. This is great if you want to chat with a friend while you play separate games.

While you are playing a game, you can view available network sessions of your party members that you can join. To display the list of sessions you can join, add the following to the `MainMenuUpdate` method:

```
// Show a party sessions the player can join
if (ButtonPressed(Buttons.Y) && !Guide.IsVisible)
    Guide.ShowPartySessions(PlayerIndex.One);
```

You can also invite the players in your party to your game session. To show the party screen from your game, add the following to the `GameLobbyUpdate` method:

```
// Show the party screen
if (ButtonPressed(Buttons.Y) && !Guide.IsVisible)
    Guide.ShowParty(PlayerIndex.One);
```

Simulating Real World Network Conditions

As expected, the conditions and speed that the network traffic performs in your game is slower than what you are currently seeing. Your connection from your PC to your Xbox 360 over your local area network is close to perfect. The physical distance is low and the amount of other traffic your packets have to deal with is relatively low. Out in the wild, Internet conditions can vary greatly. Your connection to another machine has a higher latency and you can experience packet loss.

Developers have a difficult time handling these conditions unless they could simulate them within their development environment. `NetworkSession` provides two properties that enable the developer to simulate real-world network conditions.

The `NetworkSession.SimulatedLatency` property is a `TimeSpan`, which is a simulated latency between the machines in the session. Games running on Xbox LIVE games are expected to handle latencies of 200 milliseconds while still enabling the game to be playable.

The `NetworkSession.SimulatedPacketLoss` property is a `float` value in the rage of 0 for no packet loss to 1 for 100 percent packet loss. Xbox LIVE games are expected to handle packet loss around 10 percent or a value of 0.1f.

Summary

Now you can take advantage of using Xbox LIVE and Games for Windows LIVE in your games. You learned how to create, find, and join network sessions and how to invite others to come play with you in your session.

We also covered how to simulate the real-world conditions of networks by showing you how to set the latency and packet loss for a session.

In the next chapter, you learn about dealing with media.

17

Using Media in XNA Game Studio

Using and playing media can be a great way to personalize games for the player. There are a few games completely built around using the player's media to drive game play! In this chapter, you learn everything you need to know about dealing with media, including:

- How to play songs
- How to enumerate the users media
- How to play a video
- How to use visualization data

What Is Media?

You might wonder what exactly media is, because you've spent the majority of this book working on media. You've played sounds, drawn images, what else could it be? When "media" is discussed in the context of XNA Game Studio, it means video, songs, or enumeration of items in your media library, including pictures, album art, songs, and so on.

Although the media APIs exist on every platform, they were originally designed and intended to be used on small devices. The first implementation of these APIs was in Game Studio 3.0 for the Zune device, and the full set of features are now available on Windows Phone 7, while support for these features is more limited on Xbox 360 and Windows.

Playing a Song

Playing a song is just as easy as playing other sound effects, although it does have some special characteristics. Songs are considered music, and on some platforms can be overridden by the user. For example, on Xbox 360, if a user picks a playlist and plays it via the guide in your game, regardless of what music the game plays, the music the user chooses in the guide is what he hears.

To see how easy it is to play a song, create a new game project and add the samplemusic.mp3 file to your content project. Then, declare a new `Song` variable to your game:

```
Song song;
```

In your `LoadContent` method, load the song:

```
song = Content.Load<Song>("samplemusic");
```

You're now ready to play the song. Notice that the song object doesn't have a `Play` method on it like you have seen on the sound effect classes. This is because there can be only one song playing, where you can have multiple sound effects playing simultaneously. To play the music directly after it is created in your `LoadContent` method, add the following:

```
MediaPlayer.Stop();
MediaPlayer.Play(song);
```

Use the `MediaPlayer` class to control the music. Strictly speaking, the `Stop` call isn't required, but just in case something else plays in this sample code, it is inserted directly before the `Play` method. Before you look at all the information and data you can get from the `Song` objects, let's take a quick look at the `MediaPlayer` class.

MediaPlayer

There are a few properties on the class you can use to see the current state of what is playing and how. The first one `GameHasControl` returns true if the songs you play in your game are actually played, and false if the system is currently controlling the playback of music (because the player has requested it as in the previous example). You don't have to do anything in response to this, and don't even need to check; the system does the right thing, but if you needed to know, you have that data available.

You can also detect whether the system plays audio via the `IsMuted` property. Setting this property to true mutes audio currently playing, and setting it to false unmutes it. The `IsShuffled` property has an effect only if you're playing a song list rather than a single song. When this property is true and it is time to go to the next song in the list, it randomly picks the next song from the full list of available songs in the collection rather than simply going to the next one in the list.

Use the `IsRepeating` property to detect or set whether or not the player should repeat playing from the beginning after it has finished playing its current set of music. If this is true and you play more than one song, it starts repeating only after every song in the collection has played.

If the `IsVisualizationEnabled` property returns true, calls to `GetVisualizationData` return data about the currently playing song. Visualization data can be retrieved during each frame and return a collection of frequencies and a collection of samples. A common usage of this data is to write visualizers, which are common in media players that have fancy graphics seemingly moving with the music. We discuss this more in depth later in this chapter.

Note

`IsVisualizationEnabled` always returns false on Windows Phone 7.

You can use the `PlayPosition` property to see how far into a given piece of music you are. It returns a `TimeSpan` of the duration you heard of the song so far. To find the total length of a song, use the `Duration` property of the song itself.

The `Queue` property is the current set of songs that is played by the player. It returns a `MediaQueue` object, which has three pertinent pieces of information on it: the currently active song returned from the `ActiveSong` property; the index of this song returned from the `ActiveSongIndex` property; and the total number of songs in this list, naturally returned in the `Count` property.

You can also get the `State` of the currently playing media, which returns a `MediaState` enumeration. The only values of this enumeration (and the only states available) are `Playing`, `Paused`, and `Stopped`.

The last property on the `MediaPlayer` is `Volume`, which is the volume of the music, relative to the current system volume for music. Volume is defined by decibels, where a value of 0.0f subtracts 96 decibels from the current volume, and a value of 1.0f does not modify the volume.

You've already seen the `Play` and `Stop` method, which are pretty self-explanatory. There are also `Pause` and `Resume` methods. `Pause` and `Stop` are remarkably similar, in that each of these methods stop the music. `Stop`, however, resets the current position of the music to the start, and `Pause` keeps the current position at the moment `Pause` happened. `Resume` restarts the music from the current position.

As mentioned, you can have more than one song in a queue. There are two methods to control which song you're playing, namely `MoveNext` and `MovePrevious`. Depending on the state of the `IsShuffled` property, these methods move to the next or previous song in your list, where the concept of "next" and "previous" are random.

There are also two events you can hook to be notified of status changes of the media played. You can hook the `ActiveSongChanged` event to see when the current song has changed. This can be from the `MoveNext` or `MovePrevious` methods or by the current song ending, and the system automatically moves to a new song.

You can also use the `MediaStateChanged` event to be notified when the state has changed. As mentioned, these state changes are pause, play, and stop.

Songs and Metadata

You created the song object at the beginning of this chapter, so let's take a few moments to look at what that object contains. At a high level, it is a description of a song, along with audio data necessary for the media player to play it. Most of the data is somewhat circular in nature because there are multiple ways to get to the same data.

The most direct properties for your song are the `Name` and `Duration` of the song. You can also get the `TrackNumber` and `PlayCount` if these properties exist (they are zero, otherwise). If the `IsRated` property is true, the `Rating` property includes the current rating of the song. You can also see whether the song is DRM (Digital Rights Management) protected by looking at the `IsProtected` property.

You can find out more information about the `Artist` of the song via the property of the same name, including the `Name` of the artist, a collection of `Songs` that the artist has created (including the one you got the artist object from), and a list of `Albums` the artist has created.

Of course, you can get the `Album` the song is on, and naturally that object has plenty of information, including the same `Artist` you got from the original song object (as mentioned, the references here can get quite circular). You can also get the collection of `Songs` that are on the album and the `Name` and `Duration` of the album. You can check whether the album has cover art by looking at the `HasArt` property. If this returns true, you can get the image from the `GetAlbumArt` method, which returns a `Stream` that you can use in the `Texture2D.FromStream` method to create a texture. You can also use the `GetThumbnail` method to get the art as a thumbnail.

Both the song and the album have a `Genre` property, which like the album and artist properties also includes a `Songs` property that is the collection of songs that belong to this genre. It also includes the collection of `Albums` that belong to the genre, and the `Name` of the genre itself.

For any given song, you can reference it four different ways: namely the song itself, through the album, through the artist, or through the genre. You can also get any piece of metadata that exists about the song from any of the types.

> **Note**
>
> Use the `Song.FromUri` to create a `Song` object from a Web address. This functionality doesn't work on Xbox 360 and always throws a `NotSupportedException`.

Media Enumeration

Shipping music with your game is great, but what if you want to use the media library that already exists on the user's machine? There are quite a few games out there that let the user pick music from his or her own library while playing. Naturally, this functionality is included in XNA Game Studio.

Media Library

The starting point for this functionality is the `MediaLibrary` class. Create a new Windows Phone Game project and add the following variable:

```
MediaLibrary media;
```

Initialize this object inside the `Initialize` method:

```
media = new MediaLibrary();
```

This initializes a new media library with the default source, although you can also use the overload that expects a `MediaSource` object. Using that overload forces you to create a new `MediaSource` object stating the sources type (either local or via Windows Media

Connect) and the name. You can also get the media source from the just initialized media library. Alternatively, use the `MediaSource.GetAvailableMediaSources` method to return a collection of all available media sources on your current device.

Much like the multiple ways to get to songs, the media library class is an extension of that. Start with the entire library rather than a small portion of the library (one song). It has the `Albums` property, which is the collection of albums in the library and contains everything you learned earlier. It also has the `Artists` property, the `Genres` property, and the `Songs` properties, which are the collections of these things. Just like before, you can enumerate through all of your music through any of these properties (or any combination of them). You can also use the `PlayLists` in your library, which are a collection of `Songs` that can be played. Each playlist includes a `Name` and `Duration` much like the `Albums` collection, along with the `Songs`. `Genres` property does not exist because it is a user-specified collection that includes many different genres.

The media library has a few extra things, including pictures and music. The `RootPictureAlbum` property returns a `PictureAlbum` class. Each `PictureAlbum` includes a list of `PictureAlbums` (think folders). The `Name` property is also included as in all the other objects. The `Parent` property exists, too, which is the picture album that it is a member of (except for the `RootPictureAlbum`, which has no parent). There is a `Pictures` collection on each album, too.

> **Note**
>
> On Windows and Xbox 360, the picture collection returns an empty collection.

The `Pictures` collection on the media library returns the entire list of pictures, in all albums. Each picture in the collection has a few properties, such as `Name`, `Width`, and `Height`. It also includes `Date`, which is when the picture was taken or saved, and the `Album` the picture belongs to. Much like the music, there are multiple ways to get to each picture.

Of course, enumerating the pictures isn't exciting by itself. Instead let's find a picture, modify it slightly, and then save it out. You need a device to do this, and you have two choices of how to get the picture to modify.

First, add references to Microsoft.Phone.Tasks.dll and Microsoft.Phone.dll to your project, which is where the camera task comes from (and the camera is one of the ways you get a picture). Use `#ifdef` to use the camera or a saved picture as the starting point. Assuming you want to use the camera, add the following at the top of your game1.cs code file:

```
#define USE_CAMERA
#if USE_CAMERA
using Microsoft.Phone.Tasks;
#endif
```

You need to know when you have a picture to modify, along with ways to modify them, so add a few new variables to your class:

```
bool havePicture;
SpriteFont font;
Texture2D texture;
RenderTarget2D renderTarget;
```

To add a new sprite font to your content project, call it Photo. You can keep the default values (or modify them, if you like). Now, start the task to take a picture so you can modify it. In your `LoadContent` method, add the following code:

```
    #if USE_CAMERA
    // Create the camera chooser, and save the data from it
    new CameraCaptureTask().Show();
    ChooserListener.ChooserCompleted += (s, e) =>
        {
            var args = (TaskEventArgs<PhotoResult>)e;
            // Save the picture
            Picture picture = media.SavePicture("Camera Picture" +
                DateTime.Now.ToString(), args.Result.ChosenPhoto);
            texture = Texture2D.FromStream(GraphicsDevice,
                picture.GetImage());
            havePicture = true;
        };
    #else
#endif
```

As long as the USE_CAMERA define is set, it shows the camera capture task (enabling you to take a picture), and when that has completed, you save the picture to your photo library, and then load that into the texture. Set the boolean to true to signify you're ready to modify and save it.

The other way you can get your picture is directly from your photo library. This code assumes you have pictures in the library, but does not crash if you do not. Update the `LoadContent` method to include the following in between the `#else` and the `#endif`:

```
if (media.Pictures[0] != null)
{
    texture = Texture2D.FromStream(GraphicsDevice,
        media.Pictures[0].GetImage());
}
havePicture = true;
```

Assuming you have a picture in your photo library, it now sets to the first one in the list. If you want to use this instead of the camera, simply comment out the top line (the `#define`). With a picture now set (regardless of which method you choose), you can update the code to modify it slightly and save it out. First, declare a few new variables.

You need to update the `LoadContent` method again to instantiate the other variables (you can do so after the `#endif`):

```
font = Content.Load<SpriteFont>("Photo");
renderTarget = new RenderTarget2D(GraphicsDevice,
    GraphicsDevice.Viewport.Width, GraphicsDevice.Viewport.Height);
```

This loads the font used to write across the image, creates a texture out of the picture if it exists, and creates a render target to render the modified picture to. Now, replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    if (texture != null)
    {
        // Switch to the render target so you can save it later
        GraphicsDevice.SetRenderTarget(renderTarget);
        spriteBatch.Begin();
        spriteBatch.Draw(texture, GraphicsDevice.Viewport.Bounds,
            Color.White);
        spriteBatch.DrawString(font, "I just modified\r\nthis picture!",
            Vector2.Zero, Color.White);
        spriteBatch.End();
        // Switch back to back buffer to commit the render target
        GraphicsDevice.SetRenderTarget(null);
    }
if (havePicture)
{
    SavePicture("ModifiedPicture" + DateTime.Now.ToString(),
        renderTarget);
    // Exit the game now, you're done
    Exit();
}
    base.Draw(gameTime);
}
```

This code should be familiar to you by now. Switch to your created render target, draw your original picture full screen, and write some text over it. With that out of the way, you can exit the game. Now that you have modified the image, use the following code to save the picture:

```
private void SavePicture(string name, Texture2D texture)
{
    System.IO.
MemoryStream stream = new System.IO.
➥MemoryStream();
    texture.SaveAsJpeg(stream, texture.Width, texture.Height);
    media.SavePicture(name, stream);
}
```

Of course, you need to call this method. Right before your call to `Exit` in `Draw`, add the following:

```
SavePicture("ModifiedPicture" + DateTime.Now.ToString(),
    renderTarget);
```

This is easy photo manipulation! There is also a special picture collection on the media library called `SavedPictures`, which is the collection of saved pictures. This is where the pictures from `SavePicture` end up.

Video

There are times when you might want to play a video in your game. Perhaps you have a "cut scene" in your game, or the introduction video. This functionality is easy to use and can be done in just a few lines of code.

Rendering a Video

Create a new Windows Game project and add the samplevideo.wmv file from the downloadable examples to the content project. Add a couple variables to your project to hold and control the video:

```
Video video;
VideoPlayer player;
```

Of course, you have to instantiate them in your `LoadContent` method:

```
video = Content.Load<Video>("SampleVideo");
player = new VideoPlayer();
```

Before getting into the details of these classes, let's get your video rendering. Add the following lines directly after creating the player in the `LoadContent` method:

```
player.IsLooped = true;
player.Play(video);
```

Finally, replace the `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);
    spriteBatch.Begin();
    spriteBatch.Draw(player.GetTexture(), GraphicsDevice.Viewport.Bounds,
➥Color.White);
    spriteBatch.End();
    base.Draw(gameTime);
}
```

That's it. Running the project now renders your video to the full window and repeats the video until the project is closed as in Figure 17.1.

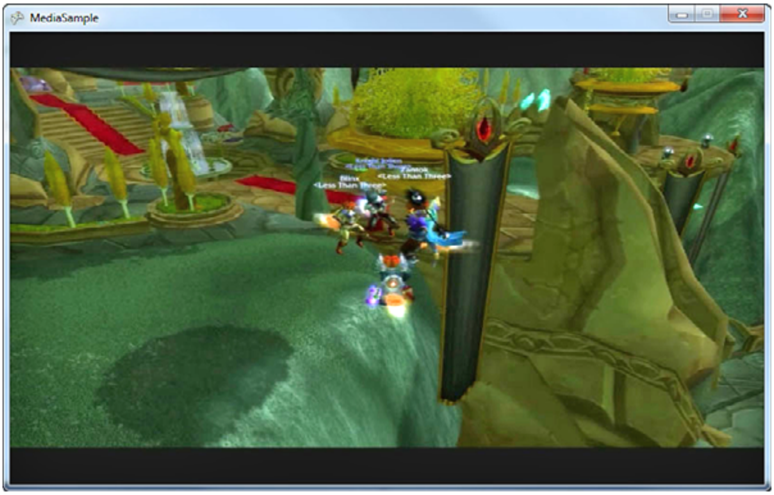


Figure 17.1 Rendered video

The `Video` object is a repository for the data about a video. It includes the `Duration` of the video and the `FramesPerSecond` (the number of frames of video in each second, if the name wasn't obvious). The native `Width` and `Height` of the video are stored here, too, along with the `VideoSoundtrackType`. The sound track type can be `Dialog`, `Music`, or `MusicAndDialog`, and these are used to determine how the system should interact with the audio from the video. If the sound track type is `Music`, and the system controls the music, the audio from your video does not play. If the setting is `Dialog`, and the system controls the music (and music is playing), the audio from the video mixes with this music. If the setting is `MusicAndDialog`, system sounds stop and the video's audio plays.

Note

The `VideoSoundTrackType` is valid only for Xbox 360.

The `VideoPlayer` object is similar to the `MediaPlayer` object from earlier. It has `IsMuted` and `IsLooped` properties (where looped is similar to repeating, except it loops only the current video). It has the `Play`, `Pause`, `Stop`, and `Resume` methods, along with the `State` property, which mirrors what you saw earlier. It also has the `PlayPosition` property you saw previously. The last method, which you used in the `Draw` method previously, is `GetTexture`. This method returns the current frame of video as a texture. So long as the video is playing (via the `Play` method), you don't need to do anything special to make it continue to update.

Because this is a Texture2D, it can be used in any place a texture is used. You can use it to render in a sprite batch, but you could just as easily use it to render onto a 3D model. Add a new model variable to your content project:

```
Model model;
```

To get the content, add the `box.fbx` model to your content project and then update `LoadContent`:

```
model = Content.Load<Model>("Box");
```

Finally, replace your `Draw` method with the following and notice your video rendering onto each face of the cube spinning around as in Figure 17.2:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    BasicEffect effect = model.Meshes[0].Effects[0] as BasicEffect;
    effect.Texture = player.GetTexture();
    effect.TextureEnabled = true;
    model.Draw(Matrix.CreateRotationX((float)gameTime.TotalGameTime.TotalSeconds) *
        Matrix.CreateRotationY((float)gameTime.TotalGameTime.TotalSeconds /2) *
        Matrix.CreateScale(3.0f),
        Matrix.CreateLookAt(new Vector3(5,0,-5),
        Vector3.Zero, Vector3.Up),
        Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
        GraphicsDevice.Viewport.AspectRatio, 1.0f, 100.0f));
    base.Draw(gameTime);
}
```

There shouldn't be anything surprising in this section of code. All you do is render a model with a slight rotation and scale, and update its texture every frame. Even though the most common use of videos is for cut scenes where they are rendered over the entire viewable area, this snippet shows it is possible to render them onto 3D objects, too.

> **Note**
>
> The `Video` and `VideoPlayer` classes are not available on Windows Phone 7. To render video here, you must use the MediaPlayerLauncher task. By using that task though, you cannot get the texture from the video to use in your game as you've done here.

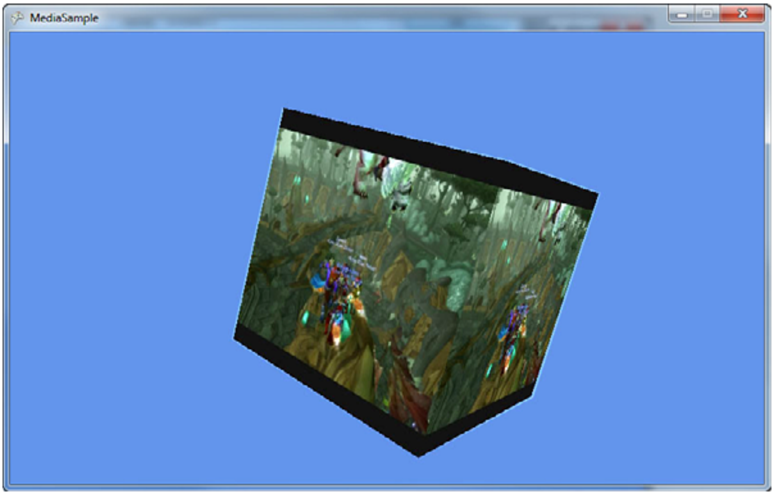


Figure 17.2 Rendered video onto a 3d cube

Visualizations

As mentioned earlier, while you are playing a song, you can get visualization data. This data is an array of frequencies and samples based on the current audio played. Although a common usage of this data is to render visualizers (and is what this code does), you can also use this data to drive your game. Frequencies are normalized between 0.0f and 1.0f, and each value represents a frequency band between 20Hz and 20KHz and can be perceived as pitch. The samples are from –1.0f to 1.0f and approximate the wave form of the sound. They can be perceived as volume.

Rendering Visualization Data

For this example though, you simply modify a texture every frame and render that to help visualize the audio played. Create a new Windows Game project, and add the samplemusic.mp3 file from the downloadable examples to your content project. Then add the following variables to your game:

```
Song song;
Texture2D texture;
VisualizationData data;
Color[] colorData;
```

Obviously, the song is the music that you play, and the texture is what you render to the screen. The visualization data is used to store the frequency and sample data, and the array of colors is used to update the texture with data from the audio. Modify your `LoadContent` method to instantiate these values:

```
song = Content.Load<Song>("SampleMusic");
data = new VisualizationData();
MediaPlayer.IsVisualizationEnabled = true;
texture = new Texture2D(GraphicsDevice,
    data.Frequencies.Count, data.Samples.Count);
colorData = new Color[texture.Width * texture.Height];
```

Loading the song should be familiar. Create the `VisualizationData` object because it creates two arrays for the data to be returned, and this is not something you want to do every frame. The color data is the same way. You also need to create the texture (and do so using the same size of data from the visualization data) and turn on the visualization data.

To update the color data every frame to see how it changes, add the following code to your `Update` method:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.A == ButtonState.Pressed)
{
    MediaPlayer.Stop();
    MediaPlayer.Play(song);
}
if (MediaPlayer.State == MediaState.Playing)
{
    MediaPlayer.GetVisualizationData(data);
}
for (int x = 0; x < data.Frequencies.Count; x++)
{
    for (int y = 0; y < data.Samples.Count; y++)
    {
        colorData[(x * data.Frequencies.Count) + y] =
            new Color(data.Frequencies[x],
                (float)Math.Asin(data.Samples[y]),
                data.Frequencies[x] + (float)Math.Asin(data.Samples[y]));
    }
}
texture.SetData<Color>(colorData);
```

You don't start playing the music until you press the A button (feel free to modify this to some other input mechanism). This gives you the chance to restart it at any time. Then, check the state of the current media, and if it is playing you get the visualization data. If the state is stopped, or if visualization data is unsupported or unavailable on this platform, the data returned is all zeros. Next, loop through all of the available data, and set each pixel in the color array to a particular color based on the sample and frequency. Finally, update the texture, and now all you need to do is render this on the screen. Replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(texture, GraphicsDevice.Viewport.Bounds, Color.White);
    spriteBatch.End();
    GraphicsDevice.Textures[0] = null;
    base.Draw(gameTime);
}
```

The only thing new you might notice is resetting the texture back to null after the sprite batch is finished. This is done because you cannot modify a texture that is set on a device, and using it in the sprite batch sets it to the device. Running your project now shows a colorful pattern on the screen when music is playing (by pressing the A button on your controller, or whatever input mechanism you specified) as seen in Figure 17.3.

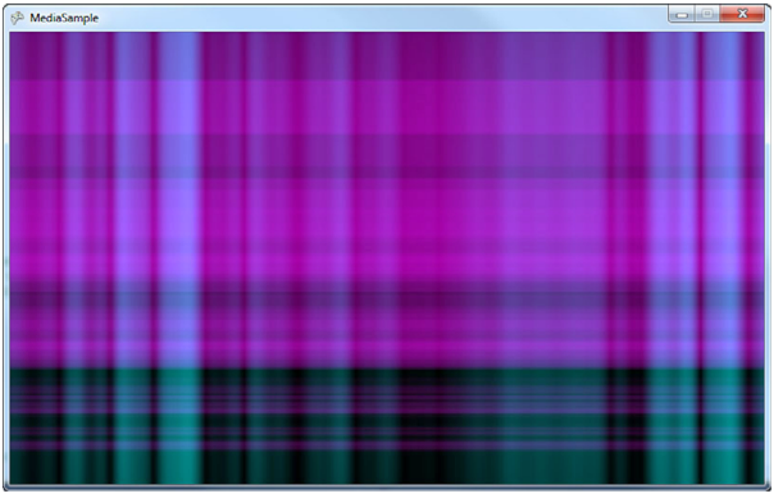


Figure 17.3 Rendered visualization data

Summary

Whew. It has been a long time coming, but we're finally done with all the features of the framework. In this chapter, you learned about the features in the Media namespace of the XNA Framework. You learned how to play music and videos, as well as enumerating through your audio and media library. You also learned how to get visualization data from your audio.

The next chapter puts it all together and takes lessons you learned throughout the rest of the book to make something awesome.

This page intentionally left blank

A

Reach vs. HiDef Chart

XNA Game Studio 4.0 introduced the `GraphicsProfile` to guarantee a set of features that a specific platform supports. In previous versions of XNA Game Studio and other graphics APIs, the developer was responsible for querying a large number of capabilities for the hardware that ran the software. The developer had to tailor their game to utilize only features that were available.

A `GraphicsProfile` enables developers to know that a specific set of features is available for a game and to focus on writing the game.

The `GraphicsProfile` is discussed more in Chapter 4, "Introduction to 3D Graphics." In this appendix you learn the differences between the `Reach` and `HiDef` profiles.

Graphics Profile Differences of Reach and HiDef

Following is a chart that defines the differences between `Reach` and `HiDef`. Although the features available for `Reach` and `HiDef` are mostly the same across the Windows, Windows Phone, and Xbox 360 platforms, some small differences are called out.

	Reach	**HiDef**
Supported platforms	Windows Phone 7 Xbox 360 Windows PCs that support at least shader model 2.0 and the other features listed in this chart.	Xbox 360 Windows PCs that support the features listed in this chart. Most DirectX 10 cards support the `HiDef` `GraphicsProfile`.
Shader model version	2.0 Windows Phone 7 does not support custom shaders.	3.0 Xbox 360 supports additional instruction not available on Windows, such as vfetch.
`Texture2D` **size maximum**	2,048	4,096

	Reach	**HiDef**
`TextureCube` **size maximum**	512	4,096
`Texture3D` **size maximum**	Not supported	256
Nonpower of two `Texture2D`	Supported as long as the following are not used: `TextureAddressMode` of Wrap, Mipmaps, or **DXT compression**	Supported
Nonpower of two `TextureCube`	Not supported	Supported
Nonpower of two `Texture3D`	Not supported	Supported
Maximum primitives per draw call	65,535	1,048,575
`IndexBuffer` format	16bit / short	32 bit / int
Vertex element formats	`Color`, `Byte4`, `Single`, `Vector2`, `Vector3`, `Vector4`, `Short2`, `Short4`, `NormalizedShort2`, and `NormalizedShort4`	`Color`, `Byte4`, `Single`, `Vector2`, `Vector3`, `Vector4`, `Short2`, `Short4`, `NormalizedShort2`, `NormalizedShort4`, `HalfVector2`, and `HalfVector4`
Texture formats	`Color`, `Bgr565`, `Bgra5551`, `Bgra4444`, `NormalizedByte2`, `NormalizedByte4`, `Dxt1`, `Dxt3`, and `Dxt5`	`Color`, `Bgr565`, `Bgra5551`, `Bgra4444`, `NormalizedByte2`, `NormalizedByte4`, `Dxt1`, `Dxt3`, `Dxt5`, `Alpha8`, `Rg32`, `Rgba64`, `Rgba1010102`, `Single`, `Vector2`, `Vector4`, `HalfSingle`, `HalfVector2`, `andHalfVector4` Floating point texture formats do not support filtering.
Vertex texture formats	Not supported	`Single`, `Vector2`, `Vector4`, `HalfSingle`, `HalfVector2`, and `HalfVector4`

	Reach	**HiDef**
Render target formats	`Color` Windows Phone: `Bgr565`, `Bgra5551`, and `Bgra4444` Xbox 360: `Rgba1010102`, `Single`, `Vector2`, `HalfVector2`, `HalfVector4`, `HdrBlendable` Windows: `HdrBlendable` and others. Call `QueryBackBufferFormat` and `QueryRenderTargetFormat`	`Color`, `HdrBlendable` Windows Phone: `Bgr565`, `Bgra5551`, and `Bgra4444` Xbox 360: `Rgba1010102`, `Single`, `Vector2`, `HalfVector2`, `HalfVector4`, and `HdrBlendable` Windows: `HdrBlendable` and others. Call `QueryBackBufferFormat` and `QueryRenderTargetFormat`.
Multiple render targets	Not supported	Supports up to four. All must share the same bit depth.
`OcclusionQuery`	Not supported	Supported
Separate alpha blend	Not supported	Supported
`Blend.SourceAlphaSaturation`	Supported with `SourceBlend` Not supported with `DestinationBlend`	Supported
Vertex streams maximum	16	16
`Stream stride maximum`	255	255

Summary

This chart should be valuable to you as you plan the features for your game. If you plan to target Windows Phone 7 or want your game to work across platforms, you should select the `Reach GraphicsProfile` and stick to its feature set. The good news is that if you set your `GraphicsProfile` as `Reach`, the XNA framework will prevent you from using features that are not available in the `Reach GraphicsProfile`, even if your development PC can support more features. This ensures your game will work on machines with lower capabilities than your own.

This page intentionally left blank

B

Using the Windows Phone FMRadio

All Windows Phone 7 devices will have a built in FM radio tuner. Unlike some other portable music devices users of Windows Phone 7 devices will be able to tune into available FM radio stations. This is very convenient when a user wants to listen to a local station for sporting events or tune to emergency broadcast stations.

The Windows Phone 7 developer APIs provide the ability for developers to utilize the built in FM Radio tuner.

In this appendix you will learn:

- How to turn the FM radio turner on
- How to set the radio region
- How to set the station frequency for the radio

Using the FMRadio in Your Windows Phone 7 Game

The Windows Phone 7 developer APIs provide an easy-to-use API that enables developers to control the FM radio tuner on the device.

The `FMRadio` type is provided in the Microsoft.Phone.dll assembly. To utilize the `FMRadio` in the sample, you first need to add the assembly to the project. Right click on the References section of your XNA Windows Phone project and select Add Reference. Click the .NET tab, select the Microsoft.Phone assembly, and then click the OK button.

`FMRadio` is defined in the `Microsoft.Devices.Radio` namespace, so you need to add the following using statement at the top of the game:

```
using Microsoft.Devices.Radio;
```

Now you need to store the instance of the `FMRadio` so you can add the following member variable to the `Game` class:

```
FMRadio radio;
```

There is only a single instance of the `FMRadio`. Because it represents the physical FM tuner built into the hardware, it does not make sense to have multiple instances that developers can set to different kinds of values. To ensure there is one and only one instance of the `FMRadio`, the `FMRadio` exposes a singleton instance through a property called `Instance`.

```
radio = FMRadio.Instance;
```

After you have a reference to this instance of the `FMRadio`, you can set the other properties exposed by `FMRadio`.

Because the FM radio band has different rules and regulations around the world, the `FMRadio` type exposes a property call `CurrentRegion` which is of the type `RadioRegion` which has values for `UnitedStates`, `Europe`, and `Japan`.

```
radio.CurrentRegion = RadioRegion.UnitedStates;
```

As you most likely know, FM radio stations utilize different frequencies. By setting the FM tuner to a specific frequency, you tune your radio into that station and can hear what broadcasts. `FMRadio` provides a `Frequency` property that enables you to read and set the current tuner frequency.

```
radio.Frequency = 101.5;
```

> **Note**
>
> You might have asked how you can test your `FMRadio` code on the Windows Phone emulator. The good news is that some fine engineers thought of this problem and decided to play some simple music on the emulator if the `FMRadio` is set to the 101.5 frequency.

The radio turner might not be turned on. To turn on the radio, you need to set the `PowerMode` property to `RadioPowerMode.On`, as shown in the following:

```
radio.PowerMode = RadioPowerMode.On;
```

With the radio turned on, you can hear audio play. If you run the sample in the emulator, you should be able to hear some simple music on a loop. If you stop the application, you might notice that the music continues to play. This is because the `FMRadio` is available throughout the Windows Phone and works like background music.

Some users might choose to disable application access to the FM radio. If the user disables this access, the `FMRadio` properties will throw a `RadioDisabledException`. Your application should handle this exception and continue to function.

Summary

The FM radio is a unique feature for Windows Phone 7 devices and provides developers with the ability to tune their games and applications into FM radio stations within their games. This appendix covered how to use the `FMRadio` type to have users' phones tune into radio stations.

This page intentionally left blank

C

Windows Phone 7 Launchers and Choosers

Windows Phone 7 has a wide variety of operations that are useful to use in your game and applications. Luckily, the system provides a large number of these to you in the form of what are called tasks. In this appendix, you learn about the tasks available on Windows Phone 7, including:

- Sending emails
- Picking contacts
- Making phone calls
- Playing video
- Using marketplace

Launchers

Conceptually, a launcher and a chooser are the same thing. The only difference is that choosers can return data back to you, whereas launchers are normally something you launch but don't expect any data back from. There are currently ten available launchers and six available choosers. Throughout this chapter, each is discussed.

Let's make a quick example to show off each of these tasks, so go ahead and create a new Windows Phone Game project. Add a new sprite font to your `Content` project named Text. Also add a movie to your project, *not* your `Content` project but the main project. You can add the `samplemovie.wmv` from the downloadable examples if you'd like. After the movie is added, change the property Copy to Output Directory to Copy if newer. Then, add the following variables to your `Game` class:

```
string _currentString = "Waiting for something...";
private Rectangle[] ButtonList = new Rectangle[16];
private const int ButtonSize = 100;

Texture2D _gray;
Texture2D _chosenPhoto;
```

The string is used to give basic status updates (and it's why you need the sprite font). Because there are 16 different tasks you can perform, the array of rectangles is used as a simple button mechanism. The `gray` texture is what you use to render your pseudo buttons, and the last texture is what you use to render the picture that the user chooses in the tasks that enable them to do so. With that, you can add the following code to your `Game` constructor:

```
graphics.PreferredBackBufferHeight = 800;
graphics.PreferredBackBufferWidth = 480;
// Use some gestures
TouchPanel.EnabledGestures = GestureType.Tap;
for (int i = 0; i < ButtonList.Length; i++)
{
    int x = (i % 4);
    int y = (i / 4);
    ButtonList[i] = new Rectangle((10 * (x + 1)) + (x * ButtonSize),
        25 + (10 * (y + 1)) + (y * ButtonSize), ButtonSize, ButtonSize);
}
```

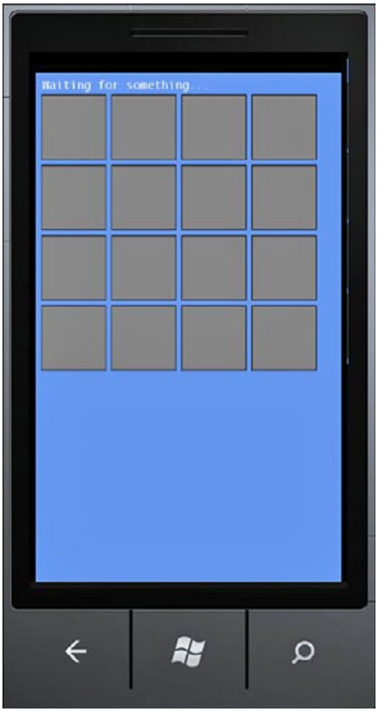
At first, you set the resolution to switch to portrait mode because many of the tasks appear that way anyway, and it makes the example look better. You will be using the `Tap` gesture to register your button clicks, so you enable that. Finally, you create your button rectangles in a 4×4 grid. You still need to create your texture and load the sprite font, though, so add the following to `LoadContent`:

```
_gray = new Texture2D(GraphicsDevice, 1, 1);
_gray.SetData<Color>(new Color[] { Color.Gray });
font = Content.Load<SpriteFont>("Text");
```

You haven't done anything with the tasks yet, but let's get the other boiler plate code so the rest of the chapter is focused solely on the tasks. Replace your `Draw` method with the following:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    foreach (Rectangle r in ButtonList)
    {
        Rectangle border = r;
        border.Inflate(1, 1);
        spriteBatch.Draw(_gray, border, Color.Black);
        spriteBatch.Draw(_gray, r, Color.White);
    }
    // Draw the chosen photo if it exists
    if (_chosenPhoto != null)
    {
        spriteBatch.DrawString(font, "Current photo:",
            new Vector2(20, 500), Color.White);
        Vector2 v = font.MeasureString("Current photo:");
        spriteBatch.Draw(_chosenPhoto, new Rectangle(20,
            (int)v.Y + 505, 200, 200), Color.White);
    }
    spriteBatch.DrawString(font, _currentString,
        new Vector2(10, 5), Color.White);
    spriteBatch.End();
}
```

First, render your gray texture into your button rectangles, but notice that you also render a slightly larger black rectangle first. The `Inflate` method increases the size of the rectangle. This gives you a small border around the pseudo buttons. Next, render the currently chosen photo if it happens to exist below all of the buttons. You then render some extra text for basic status updates, and you've now got your basic framework setup and can start learning about the tasks. Running the application now would show you something similar to Figure C.1.



Waiting for something...

Figure C.1 The basic framework for the tasks example

You'll need a helper function to detect which button you've pressed, so add that now:

```
int GetButton(GestureSample gesture)
{
    for (int i = 0; i < ButtonList.Length; i++)
    {
        if (ButtonList[i].Contains((int)gesture.Position.X,
            (int)gesture.Position.Y))
        {
            return i;
        }
    }
    return -1;
}
```

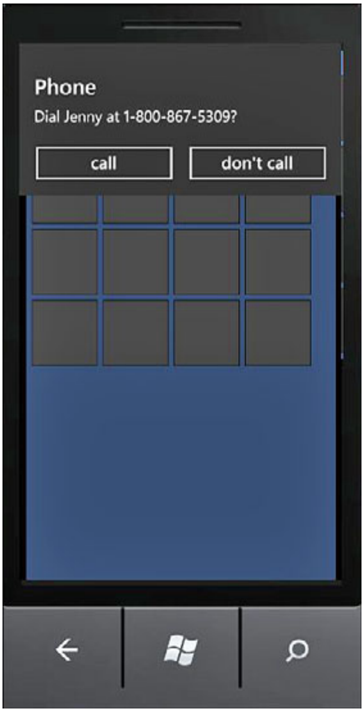
This helper goes through each button rectangle in your array, and if the position of the gesture you've passed in is contained within that rectangle, it returns its index into the array. This enables you to detect individual button presses and react to them. Now, it's time to start looking at the actual tasks. Start with the launchers, as they do not have any need to return data. Add the following to the `Update` method:

```
while (TouchPanel.IsGestureAvailable)
{
    GestureSample gesture = TouchPanel.ReadGesture();
    switch (GetButton(gesture))
    {
        case 0: // Phone caller
            {
                PhoneCallTask phone = new PhoneCallTask();
                phone.DisplayName = "Jenny";
                phone.PhoneNumber = "1-800-867-5309";
                phone.Show();
            }
            break;
        default:
            // Do nothing
            break;
    }
}
```

Simply look for the Tap gestures, and if you find one, see if it pressed a button (although you're currently only looking for one button). If something else was tapped, you simply ignore it. Because this is a phone after all, one of the first tasks you might want to do is to have your game ask the player to make a phone call. This could be trash talking at its finest! Notice a similar pattern to all of the launchers. After you create the object, you might set a few properties, and then call `Show`. In this case, you can set the

`DisplayName` and the `PhoneNumber` properties. You get this information from another task later when we discuss choosers.

Note that simply invoking this launcher won't make a phone call, however. The user has to actually initiate that action, so when you run this now, it simply asks the user if he's like to call the person specified, much like you see in Figure C.2.



Phone

Dial Jenny at 1-800-867-5309?

Figure C.2 Making a phone call

After your player has called someone to gloat over his victory, perhaps you'd like to send him to your web site? Hook up the web launcher to the second button by adding the following `case` statement:

```
case 1: // Web task
    {
        WebBrowserTask web = new WebBrowserTask();
        web.URL = "http://developer.windowsphone.com";
        web.Show();
    }
    break;
```

The only property here is the `URL` that you want to navigate to. This launches the web browser and navigates to the specified `URL`. However, maybe you want your game players to tell all their friends how awesome your game is. You can have them send an email by using the email launcher. Add the following case statement for the Next button:

```
case 2: // Send email
    {
        EmailComposeTask email = new EmailComposeTask();
        email.Subject = "This is my game!";
        email.Body = "This game rocks.";
        email.Show();
    }
    break;
```

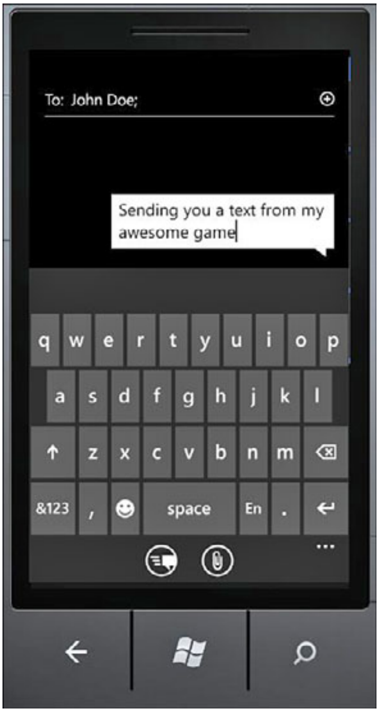
In the example, you use two of the four properties available: the `Subject` and `Body`. These are used to autopopulate the text of the email message. You can also set the `To` and `Cc` properties to have recipients already in the email as it launches. Of course, even though email is nice, this is a mobile phone, and sending a text message is much more modern. You can do this by adding the following case statement:

```
case 3: // SMS Compose task
    {
        SmsComposeTask sms = new SmsComposeTask();
        sms.Body = "Sending you a text from my awesome game";
        sms.To = "John Doe";
        sms.Show();
    }
    break;
```

Much like the earlier tasks, the `Body` is the content of the message, where the `To` property dictates the recipient. Just like the phone-calling task, invoking this doesn't send the message directly without the user initiating it either, as seen in Figure C.3.

The next four tasks are related to the marketplace. Because you want to ensure your games and applications have the best marketplace visibility, these tasks can help with that, and they can even direct people to your other applications! First is the details page, so add the following case statement:

```
case 4: // MarketplaceDetailTask Task
    {
        MarketplaceDetailTask market = new MarketplaceDetailTask();
        market.ContentType = MarketplaceContentType.Applications;
        market.Show();
    }
    break;
```



To: John Doe;

Sending you a text from my awesome game

Figure C.3 Sending a text message

The details launcher includes two properties: the `ContentType`, which can be either `Applications` or `Music`, and the `ContentId`, which is the unique identifier for a piece of content on the marketplace. This enables you to go to the detail page of any game you'd like, although by default, it goes to your own. You should make the Next button go to a particular hub, so add the following case statement:

```
case 5: // MarketplaceHubTask Task
    {
        MarketplaceHubTask market = new MarketplaceHubTask();
        market.ContentType = MarketplaceContentType.Applications;
        market.Show();
    }
    break;
```

The only property here is to choose which hub you want to go to by picking the `ContentType`, and you can choose only `Applications` or `Music`, as before. Picking `Applications` takes you to that hub, much like you see in Figure C.4.

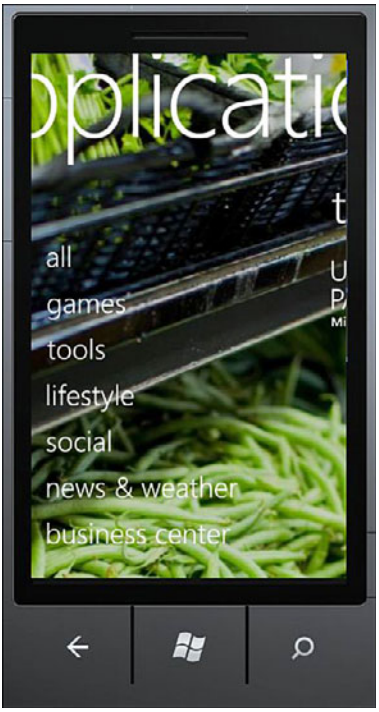


Figure C.4 Visiting a marketplace hub

The marketplace also enables users to submit reviews for the games and applications that they play. You can use the task in the next case statement to enable the user to go to the review page (and even submit his own) right from your game:

```
case 6: // MarketplaceReviewTask Task
    {
        MarketplaceReviewTask market = new MarketplaceReviewTask();
        market.Show();
    }
    break;
```

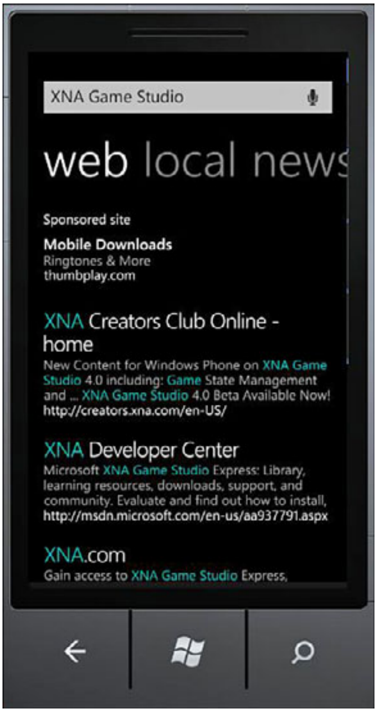
Of course, you might want to send them to the marketplace to search for other games you've made! In this case, you use the following task found in this case statement:

```
case 7: // MarketplaceSearchTask Task
    {
        MarketplaceSearchTask market = new MarketplaceSearchTask();
        market.ContentType = MarketplaceContentType.Applications;
        market.SearchTerms = "xna";
        market.Show();
    }
    break;
```

Once again, you can choose the `ContentType` you are interested in, and this time you can also include the `SearchTerms` you want to look for. If you only want results from things you care about (say your games), make sure your search is specific enough to find only those! Perhaps you have a unique developer name or something. Searching through the marketplace is nice, but what if you want to enable users to search anywhere? You can assign that task to the Next button with this case statement:

```
case 8: // Search Task
    {
        SearchTask search = new SearchTask();
        search.SearchQuery = "XNA Game Studio";
        search.Show();
    }
    break;
```

The only property here is the `SearchQuery`, which is what the search box is prepopulated with (and the initial search results are based on). This is the same as hitting the Search button on the actual phone itself and looks like what you see in Figure C.5.



Sponsored site

Mobile Downloads
Ringtones & More
thumbplay.com

XNA Creators Club Online - home
New Content for Windows Phone on XNA Game Studio 4.0 including: Game State Management and ... XNA Game Studio 4.0 Beta Available Now!
http://creators.xna.com/en-US/

XNA Developer Center
Microsoft XNA Game Studio Express: Library, learning resources, downloads, support, and community. Evaluate and find out how to install,
http://msdn.microsoft.com/en-us/aa937791.aspx

XNA.com
Gain access to XNA Game Studio Express,

Figure C.5 Searching

The last of the launchers (tasks without any return values) is the one to host media and it is the reason you added the movie to your project when you first created it at the beginning of this chapter. You can hook that up to the Next button with the following case statement:

```
case 9: // MediaPlayerLauncher Task
    {
        MediaPlayerLauncher media = new MediaPlayerLauncher();
        media.Controls = MediaPlaybackControls.All;
        media.Location = MediaLocationType.Install;
        media.Media = new Uri("samplemovie.wmv", UriKind.Relative);
        media.Show();
    }
    break;
```

You probably have more control with this task than any of the others. First, you can control explicitly which portions of the playback controls you wish to allow the user to have access to via the `Controls` property. This can range from `None`, to `All`, to everything in between. Next, you need to inform the launcher where the media will comes from by the `Location` property. You can choose `None` if your media is somewhere external to the phone (say a website), whereas you choose `Data` if the media is stored within your application's isolated storage file. If the media is part of your package (like it is in this example), choose the `Install` option. Finally, you need to pass in a `URI` to the actual `Media` before calling the `Show` method.

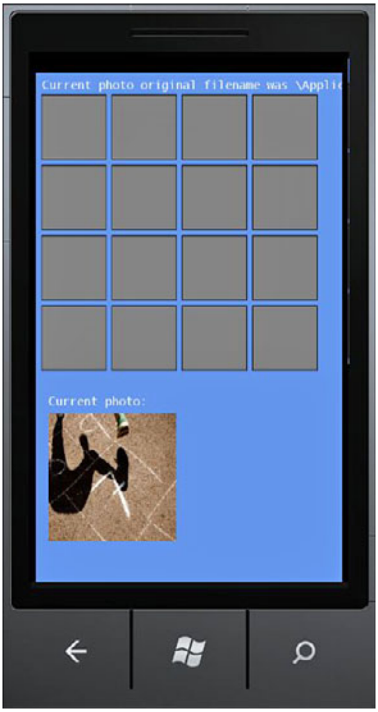
Choosers

The rest of the tasks all have a mechanism in which they can return data back to you. They each do it through the same way—a `Completed` event that is fired. The first task you look at is the photo chooser, so add the following case statement:

```
case 10: // Photochooser Task
    {
        PhotoChooserTask photo = new PhotoChooserTask();
        photo.ShowCamera = true;
        photo.Completed += new EventHandler<PhotoResult>((o, result) =>
        {
            if (result.TaskResult == TaskResult.OK)
            {
                _currentString = "Current photo original filename was " +
                    result.OriginalFileName;
                _chosenPhoto = Texture2D.FromStream(GraphicsDevice,
                    result.ChosenPhoto);
            }
            else if (result.TaskResult == TaskResult.Cancel)
            {
                _currentString = "Choosing photo was cancelled";
        }
    });
    photo.Show();
}
break;
```

There are a few properties you can set here before the `Show` method. The `ShowCamera` property enables you to specify whether you allow a new picture to be taken and returned (by passing in `true`), or if it must already exist in the picture list (by passing in `false`). You can use the `PictureWidth` and `PictureHeight` properties to specify the maximum size of the image (it is cropped beyond these limits). Finally, you have the `Completed` event.

In this event, if the `TaskResult` is `OK`, then use the `ChosenPhoto` property to create the `_chosenPhoto` object that is rendered in your scene as you see in Figure C.6. If the user cancels the task without choosing a photo, update the status text signifying that. You also use the `OriginalFileName` property in the results in your status text.



Current photo original filename was \Applic

Current photo:

Figure C.6 Rendering after picking a photo

The next task is remarkably similar, except it only enables you to get the picture from the camera, rather than the stored pictures. Using the following case statement for the Next button shows this:

```
case 11: // CameraCaptureTask Task
    {
        CameraCaptureTask camera = new CameraCaptureTask();
        camera.Completed += new EventHandler<PhotoResult>((o, result) =>
        {
            if (result.TaskResult == TaskResult.OK)
            {
               _currentString = "Camera pictures have no original filename " +
                    result.OriginalFileName;
                _chosenPhoto = Texture2D.FromStream(GraphicsDevice,
                    result.ChosenPhoto);
            }
            else if (result.TaskResult == TaskResult.Cancel)
            {
                _currentString = "Taking picture was cancelled";
            }
        });
        camera.Show();
    }
    break;
```

The next two tasks are both related to email addresses and can give you the information you needed earlier to use in other tasks. The first one lets you choose a contact and returns that contact's email address. Add the following case statement:

```
case 12: // Email Chooser
    {
        EmailAddressChooserTask email = new EmailAddressChooserTask();
        email.Completed += new EventHandler<EmailResult>((o, result) =>
        {
            if (result.TaskResult == TaskResult.OK)
            {
                _currentString = "Email address is " + result.Email;
            }
            else if (result.TaskResult == TaskResult.Cancel)
            {
                _currentString = "Choosing email was cancelled";
            }
        });
        email.Show();
    }
    break;
```

If the user chooses a contact, you print the `Email` address as part of the status text. If you want to allow them to add a new contact based on an `Email` address instead, you can do that with the following:

```
case 13: // Save email task
    {
        SaveEmailAddressTask save = new SaveEmailAddressTask();
        save.Email = "John.Doe@fakecompany.org";
        save.Completed += new EventHandler<TaskEventArgs>((o, result) =>
        {
            if (result.TaskResult == TaskResult.OK)
            {
                _currentString = "Successfully saved contact email.";
            }
            else if (result.TaskResult == TaskResult.Cancel)
            {
                _currentString = "Cancelled saving contact email.";
            }
        });
        save.Show();
    }
    break;
```

Here, you specify the `Email` address you want to create the contact for before calling `Show`. The next two tasks are virtually identical to these two, just with phone numbers replacing emails. You can see these by using the following:

```
case 14: // Phone Number Chooser
    {
        PhoneNumberChooserTask phone = new PhoneNumberChooserTask();
        phone.Completed += new EventHandler<PhoneNumberResult>((o, result) =>
        {
            if (result.TaskResult == TaskResult.OK)
            {
                _currentString = "Phone number is " + result.PhoneNumber;
            }
            else if (result.TaskResult == TaskResult.Cancel)
            {
                _currentString = "Choosing phone number was cancelled";
            }
        });
        phone.Show();
    }
    break;
case 15: // Save phone number task
    {
        SavePhoneNumberTask save = new SavePhoneNumberTask();
        save.PhoneNumber = "1-800-867-5309";
        save.Completed += new EventHandler<TaskEventArgs>((o, result) =>
        {
            if (result.TaskResult == TaskResult.OK)
            {
                _currentString = "Successfully saved contact email.";
            }
            else if (result.TaskResult == TaskResult.Cancel)
            {
                _currentString = "Cancelled saving contact email.";
            }
        });
        save.Show();
    }
    break;
```

That completes the list of built-in tasks that ship with Windows Phone 7.

Tasks That Cause Tombstoning

Tombstoning is discussed in depth in Appendix D, "Dealing with Tombstoning," but because some of these tasks can cause Tombstoning, it should be mentioned here. The following tasks can cause your application to be tombstoned:

- PhoneCallTask (only if advanced options are used)
- WebBrowserTask
- EmailComposeTask
- SmsComposeTask
- MarketplaceDetailTask
- MarketplaceHubTask
- MarketplaceReviewTask
- MarketplaceSearchTask
- SearchTask
- SaveEmailAddressTask
- SavePhoneNumberTask

Notice that this is 11 of the 16 available tasks. It just goes to show your application needs to handle tombstoning correctly!

Summary

Many common operations that the Windows Phone platform supports are exposed to application and game developers via tasks. This appendix has covered each of these tasks in depth, so you now have the knowledge to use them in your games!

This page intentionally left blank

D

Dealing with Tombstoning

On Windows Phone 7, the system controls the lifetime of your application. When your game isn't in the foreground, the system can (and almost certainly will) stop the process. In this appendix, we cover:

- What tombstoning is
- How to handle your application that is tombstoned

What Is Tombstoning?

At the highest level, an application can have two states: It can either be running or it can be not running. Windows Phone 7 adds an extra state to the case where the application is not running. It can either be closed or it can be tombstoned. Tombstoning is when the process for the application is killed because it is no longer in the foreground, but the system remembers it because it might need to go back to that application.

For example, say your application is running, and the user clicks the home screen. Your application is no longer in the foreground, and the process is killed. It is tombstoned, but the system needs to remember it, because if the user clicks the Back button, your application needs to be shown again.

This concept is important enough that it bears repeating. If your application is not in the foreground, then it is not running. This can happen if the user clicks the home screen or if something as simple as your application launching one of the launchers or choosers available in Windows Phone 7 occurs.

Now, games and applications exit all the time, so this shouldn't be a surprise to anyone; however, the difference is that the user's perception should be that the game hasn't exited. For example, while the user plays your game, she makes a phone call. She clicks the Windows button on the phone to get to the home screen, clicks the Phone button, and makes the call. Then she clicks the Back button to get back to the home screen, and then again to get back to the game.

From the user's perspective, she should be in the same position now as she was before the phone call. She never quit the game; she simply navigated away and is now back. From the game's perspective though, it was killed right after the user clicked the home screen and it was no longer in the foreground. This implies that your game needs to be smart enough to save its state before it's killed, and reload that state when it starts up again after it is tombstoned.

Reacting to Tombstoning

For an easy way to detect a killed process, you can use a simple application. To do this, open the `IsolatedStorage` example you created in Chapter 14, "Storage." Add the following variable to your game:

```
int currentFrameCount = 0;
```

Use a simple frame counter to detect where you are in the game, so that you can restore it after the application is killed by tombstoning. To increment it every frame, add the following variable to your `Update` method:

```
currentFrameCount++;
```

To draw this counter, add the following to your `Draw` method before the `spriteBatch.End` call:

```
spriteBatch.DrawString(font, currentFrameCount.ToString(),Vector2.Zero,
➥Color.Black);
spriteBatch.DrawString(font, currentFrameCount.ToString(), Vector2.One,
➥Color.White);
```

You don't actually need two calls, but having the white text slightly offset from the black text gives it a nice contrast to the random small x that might be rendered everywhere. Every time you run the game, the frame counter starts at zero and begins incrementing, rather than starting where you left off if it was tombstoned.

Those who are familiar with the previous version of XNA Game Studio, and who want to easily make the game portable to other platforms that XNA supports, probably want to use the available events, and actually, in this example so far you already are. So, in your `OnExiting` override, add the following after you create the `IsolatedStorageFile` object:

```
using (BinaryWriter writer = new BinaryWriter(file.CreateFile("currentState")))
{
    writer.Write(currentFrameCount);
}
```

This is similar to what you already did in that method. To read the frame counter back again, add the following to your `OnActivating` event after the creation of the `IsolatedStorageFile` object:

```
using (BinaryReader reader = new BinaryReader(file.OpenFile("currentState",
    FileMode.OpenOrCreate)))
{
    if (reader.BaseStream.Length > 0)
        currentFrameCount = reader.ReadInt32();
}
```

Run the game, let the frame counter increment some, and then click the Back button to exit the game. Restart the game and notice that like before, your data is loaded up at startup and your counter continues where it left off. Wait a minute and when you click the Back button, notice that you exit the game. You don't need to save your state because it is exiting.

Now, unlike a Silverlight application, by default an XNA application does not exit if you press the Back button while the game runs. It exits only when you call the `Game.Exit` method (aside from when it tombstones, of course). You might not notice this because the default template for the new project includes this functionality. Notice it at the beginning of your `Update` method. You can use this knowledge to control when you save the data. Add a simple flag variable to know when you exit:

```
bool isExiting = false;
```

You can then modify your check for the Back button in the `Update` method to set the following value:

```
// Allows the game to exit
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
{
    isExiting = true;
    this.Exit();
}
```

Now you know the difference in the `OnExiting` override between the system killing your process and the user actually quitting the game. So, update the code with the following to handle this situation:

```
if (isExiting)
{
    file.DeleteFile("currentState");
}
else
{
    using (BinaryWriter writer = new
BinaryWriter(file.CreateFile("currentState")))
    {
        writer.Write(currentFrameCount);
    }
}
```

The big difference is that if you actually exit the game, you delete the file that stores the current state. You can do something different—for example, you can leave the code as it was and replace the single `write` call with the following:

```
writer.Write(isExiting ? 0 : currentFrameCount);
```

This works in this scenario, but is difficult in more complex situations. Sometimes, it's easier to delete the file. Because the reading code is already handling the case of the file not existing, this works great. Now, if you run the game, let the frame counter get to a few hundred, click the Back button, and then run the game again, it restarts from zero. On the other hand, if you run the game, let the counter get to a few hundred, click the Home button, and then go back to the application, it restores the counter where it was!

Using Phone-Specific Features to Handle Tombstoning

Because XNA Game Studio is a cross platform API, it doesn't always map exactly one to one to what the system is doing. For example, you get both the `Deactivated` and the `Exiting` event from the `Game` object if you either exit manually or are tombstoned. You also get the `Activated` event when you start up regardless of how you started. This makes it easy to write code that works on all platforms, but loses some of the nuances that a particular platform exposes.

Luckily, you can also use features specifically for the phone in order to do the same thing with more clarity about what is going on behind the scenes! First, add two new references to your project: one for `Microsoft.Phone` and the other for `System.Windows`. The APIs you need are in the first assembly, but they have a dependency on the second assembly and the system complains if you don't add the second reference. You'll also want to add another using statement to the top of your project:

```
using Microsoft.Phone.Shell;
```

Now, delete the code you added so far (or comment it out, because you will add similar code later). This time, you use the events the phone itself fires to determine which pieces of code to use. The phone fires four different events depending on the situation.

The `Launching` event occurs when the application launches for the first time, but does not occur when it comes back to life after it is tombstoned. The `Activated` event fires when the application comes back to life from being tombstoned, but not when it is launched the first time. The `Deactivated` event fires when you are about to be tombstoned, but not when you actually exit. Finally, the `Closing` event happens when the application actually exits, but not when it is tombstoned.

Armed with that knowledge, you can update your code to handle all of the scenarios required! First, because you want to save and load the point data in all scenarios, add the following two helper functions that you can use later.

```
void LoadPointData(IsolatedStorageFile file)
{
    using (BinaryReader reader = new BinaryReader(file.OpenFile("pointData",
        FileMode.OpenOrCreate)))
    {
        for (int i = 0; i < reader.BaseStream.Length / 8; i++)
        {
            points.Add(new Vector2(reader.ReadSingle(), reader.ReadSingle()));
        }
    }
}
void SavePointData(IsolatedStorageFile file)
{
   using (BinaryWriter writer = new BinaryWriter(file.CreateFile("currentState")))
    {
        writer.Write(currentFrameCount);
    }
}
```

This is the same functionality as before, but encapsulated in a method. So that you don't need to create an `IsolatedStorageFile` object every time, add a new variable to your game:

```
IsolatedStorageFile file;
```

To implement the tombstoning scenario using the phone events, add the following code to the end of your game constructor:

```
file = IsolatedStorageFile.GetUserStoreForApplication();
PhoneApplicationService.Current.Launching += new EventHandler<LaunchingEventArgs>(
    (o, e) =>
    {
        // We got launched
        LoadPointData(file);
    }
    );
PhoneApplicationService.Current.Activated += new EventHandler<ActivatedEventArgs>(
    (o, e) =>
   {
        // We got activated
        LoadPointData(file);
       using (BinaryReader reader = new BinaryReader(file.OpenFile("currentState",
            FileMode.OpenOrCreate)))
        {
            if (reader.BaseStream.Length > 0)
                currentFrameCount = reader.ReadInt32();
        }
    }
    );
PhoneApplicationService.Current.Deactivated += new EventHandler
➥<DeactivatedEventArgs>(
    (o, e) =>
    {
        // We got tombstoned
        SavePointData(file);
        using (BinaryWriter writer = new BinaryWriter(file.CreateFile
➥("currentState")))
        {
            writer.Write(currentFrameCount);
        }
    }
    );
PhoneApplicationService.Current.Closing += new EventHandler<ClosingEventArgs>(
    (o, e) =>
    {
        // We are exiting the application
        SavePointData(file);
        file.DeleteFile("currentState");
    }
    );
```

Now, you have fully handled all of the tombstoning scenarios. After creating the isolated storage file and handling the `Launching` event, all you need to do is load the point data. You don't need to load your current frame counter because you won't come back from being tombstoned! Next, you handle the `Activated` event, and here you again load the point data, but you also load the current frame counter because you're coming back from being tombstoned.

Next, handle the `Deactivated` event, and save both the point data and the current frame counter because you're being tombstoned. Finally, handle the `Closing` event, and because you are not being tombstoned, delete your current frame counter and save the point data.

Obviously, in a real game, the state you would need to save while being tombstoned would be more complicated, but you now should be able to save what you need at the appropriate times. You can never predict when your game will be tombstoned, so you need to handle it correctly.

Handling Graphics Resources During Tombstoning

The most difficult tombstoning task to grasp is how to reload content. As mentioned before, because the process is killed, everything needs to be reloaded. What's worse, if your application doesn't begin responding quick enough when activated after tombstoning, it could be completely killed or the user could be sent back to the home screen.

So, for example, if you keep all of the content loading in the `LoadContent` overload, but there is the content takes 10 or 15 seconds to load completely, your application may be completely killed, and the user is sent back to the home screen. This means you need to keep the reactivation code path fast, but also keep the user informed about what's going on. The system displays a Resuming screen when it first starts reactivating the application, but after that, it is up to you.

If you have little content, you can load it all in the `LoadContent` menu. If you have content that might take a while to load, you only want a minor splash screen type of content to be loaded with the rest of your content loading after the activation has occurred. After the first `Update` and `Draw` methods have been called, your application is restored from tombstoning and you are free to load whatever content you like, although a good user experience shows the user progress if it takes a long time.

Summary

Tombstoning is one of the most important features to understand for Windows Phone 7 applications. It helps distinguish the great applications from the merely good (or even bad) ones. You need to do the work to ensure your applications handle it well.

This page intentionally left blank

Index

Numbers

2D avatars, render targets, 263-265

2D graphics
- coordinate system, 13
- drawing to screen, 14–16
  - animation, 20–21
  - controlling state, 21–25
  - Draw method, 16–19
  - moving things around, 19–20
- explained, 13–14
- GraphicsDeviceManager, 14
- SpriteBatch object, 16
  - animation, 20–21
  - controlling state, 21–25
  - drawing, 16–19
  - moving things around, 19–20
- text rendering, 25–27
- Texture2D class, 14

2D sprites, 43

3D audio positioning, 356-357

3D graphics, 41-43
- coordinate systems, 44–45
- defined, 42
- matrix, 53–54
  - combining matrix transforms, 58
  - identity, 54–55
  - manipulating vectors, 59, 61
  - rotation, 56–57

    scale, 57
    translation, 55
  vectors, 46
    addition, 47-48
    cross product, 50
    dot product, 49
    negation, 49
    point versus direction and magnitude, 46
    scalar multiplication, 48
    subtraction, 48
    Vector4, 46
    XNA Game Studio, 51, 53
3D math, 43
3D rendering
  drawing primitives, 71-72
    DrawIndexedPrimitives, 80, 82
    DrawPrimitives, 78-80
    DrawUserIndexedPrimitives, 75-78
    DrawUserPrimitives, 72-75
    primitive types, 71
    vertex types, 71
  GraphicsAdapter class, 67-68
  GraphicsDevice, 69-70
    creating, 70
    reference devices, 71

A

acceleration data
  accelerometer, 344-346
  autorotation and, 348
  reading, 344
accelerometer, 344-346
  moving sprites, 346-347
AccelerometerSensor, 344
accessing AvatarDescription, 240-241
adding
  custom animation to games, 283-284
  SoundEffectInstance, 357-360
  wave files, XACT, 361, 363-364
AllowOnlineCommunication, 407
AllowOnlineSessions, 407
AllowPremiumContent, 407
AllowPurchaseContent, 407
AllowTradeContent, 407
AllowUserGeneratedContent, 407
alpha blending, effect states, 211-213
AlphaTestEffect, 105, 124, 309
ambient light, 108, 186-190
analog input, 312
angles, converting to radians, 19
animated model properties, 138
animation
  2D graphics, 20-21
  avatars, 243-244
    blending, 253-257, 259-260
    functionality through interfaces, 246
    playing multiple, 249-252
    transforms and expressions, 245
    updating, 244
  custom animations
    avatars, 265-273
    content processors, creating, 273-282
AnimationBlender, 254, 257, 259
API, storage, 387-388
App Hub membership, installing XNA Game Studio 4.0, 6-7
Apply3D, 360
ApplyChanges, 31
aspect ratio, 90
AspectRatio, 90

assets, content pipeline, 226-234

AsyncCallback, 385

attenuation, 209

audio playback
  looping with SoundEffect, 356
  using cue, 364-366

AudioEmitter, 356

AudioEngine, 365

AudioListener, 357

autorotation, accerlation data and, 348

AvailableNetworkSession, 433

AvatarAnimation, 243-244, 250
  functionality through interfaces, 246
  transforms and expressions, 245
  updating, 244

AvatarAnimationPreset, 244

AvatarCustomAnimation, 270, 285

AvatarDescription
  accessing avatar information, 240-241
  Changed event, 243
  constructing from byte arrays, 242
  creating random, 242

AvatarExpression, 270

AvatarEye, 245

AvatarEyebrow, 245

AvatarMouth, 245

AvatarRenderer, 246-247
  deterring current state of, 248
  drawing while loading, 247-248

avatars, 239-240
  2D avatars, render targets, 263-265
  accessing information using AvatarDescription, 240-241
    Changed event, 243
    constructing AvatarDescription, 242
    creating random, 242
  animations, 243-244
    blending, 253-257, 259-260
    functionality through interfaces, 246
    playing multiple, 249-252
    transforms and expressions, 245
    updating, 244
  custom animations, 265-273
    adding to games, 283-284
    content processors, 273-282
    updating games to use, 284-285
  drawing with AvatarRenderer, 246-247
    deterring current state of, 248
    while loading, 247-248
  interacting with objects, 260-263
  modifying lighting, 248-249

B

backbuffer, 65

backface culling, 63

base.Draw method, 303

BasicEffect, 105-107, 307
  effect interfaces, 121-122
  fog, 119-121
  lighting, 108-114
  textures, 114-115, 118-119
  vertex colors, 115-118

Begin method, 22

BeginMark, 299

BeginShowKeyboardInput, 316, 318-320

BeginShowMessageBox, 396

BeginShowSelector method, 387

BinaryWriter helper class, 379

BlendFactor, 147

blending, 65
  animations for avatars, 253-260
BlendState object, 142-148
  premultiplied alpha, 148-149
Blinn-Phong shading, specular lighting, 200-202
bones, 98-99
BouncingImage class, 38
build systems, tracing content, 215-216
BuildAndLoadAsset method, 232, 237
built-in shaders, cost of, 307-308
ButtonPressed method, 415
byte arrays, constructing AvatarDescription, 242

C

calculating lighting, 110
cameras, 85-86
  static cameras, 94-95
Changed event (AvatarDescription), 243
chatpad input, 313
chatpads, input, 313
choosers, Windows Phone 7, 472-476
choosing
  content processors, 220
  storage devices, 385
Clamp, 168
CleanSkeleton method, 280
ClearOptions, 153
CLR (Common Language Runtime), 2
CLR profiler, 306
color, vertex, 179-180
color value, 65
combining
  content, 226-234
  matrix transforms, 58
Common Language Runtime (CLR), 2
CompareFunction.LessEqual, 151
components, GameComponents, 38-40
connections, Xbox 360 gamepad, 333
constructing AvatarDescription from byte arrays, 242
containers, storage, 382-383
content, tracing through build system, 215-216
content importers, 223-226
Content object, 15
content pipeline, 216
  combining and building assets, 226-234
  combining what you have learned so far, 235-237
  content importers, 223-226
  content processors, 216-222
  extensions, debugging, 222
  tracing content, 215
content processors, 216-222
  choosing, 220
  creating, 273-276, 278-282
  properties, 221
ContentImporter, 230, 233
ContentManager, 237
controllers, Xbox 360 gamepad, 332-333
converting angles to radians, 19
coordinate systems, 13
  3D graphics, 44-45
CopyAbsoluteBoneTransformsTo, 99
CopyBoneTransformsFrom method, 99
CopyBoneTransformsTo method, 99
cost, 288
  of AlphaTestEffect, 309
  of built-in shaders, 307-308
  of DualTextureEffect, 308
  of EnvironmentMapEffect object, 308
  of SkinnedEffect object, 308

CPU bound, 294
CreateFile method, 378
CreateLookAt, 86, 88
CreateOrthographic, 93
CreateOrthographicOffCenter, 93
CreatePerspective method, 92
CreatePerspectiveFieldOfView, 86, 89
CreatePerspectiveOffCenter method, 92
CreateSessionDraw, 417
CreateSessionUpdate, 424
cross product, vectors, 50
cue, audio playback, 364-366
culling, backface, 63
CullMode, 166
CurrentPosition, 255, 272
custom animations
- adding to games, 283–284
- avatars, 265–273
  - content processors, 273–282
- updating games to use, 284–285

custom effects, 171
- creating, 172–173
- drawing with, 177–178
- effect states, 209–210
  - alpha blending, 211–213
- HLSL (High Level Shading Language), 172
- lighting, 186
  - ambient lighting, 186–190
  - diffuse lighting, 192–197
  - emissive lighting, 198
  - fog, 202–206
  - point lights, 206–209
  - specular lighting, 199–202
  - triangle normals, 190–192
- parts of effect files, 173
  - global variables, 174
  - vertex structures, 174–177
- texturing, 180–183
  - repeating textures, 184–186
  - setting sampler states, 183–184
- vertex color, 179–180

cutouts, depth, 153

D

DancePad, 332
data, isolated storage, saving and loading, 377-379
dead zones, thumb sticks (Xbox 360 gamepad), 332
DebugCommandUI, 306
debugging content pipeline extensions, 222
DebugManager, 306
DefaultProcessor, 231
depth, cutouts, 153
depth buffer, faking shadows, 158-161
depth test, 65
DepthBufferEnable property, 154
DepthBufferFunction, 154
DepthBufferWriteEnable, 154
DepthStencilState, 149-155
deterring current state of AvatarRenderer, 248
device states, 141-142
- BlendState object, 142–148
  - premultiplied alpha, 148–149
- DepthStencilState, 149–155
- RasterizerState, 164–166
- stencil buffer, 161, 163–164

devices
  getting, 383–386
  storage, 382–383
    choosing, 385
diffuse lighting, 192-196
  multiple lights, 196–197
  oversaturation, 197
digital input, 312
direction versus point (vectors), 46
directional diffuse lighting, 196
directional lighting, 109-111, 193
DirectX, 2
DiscardContents, 156
displaying GestureSample data, 341-342
dot product, 49
downloading tools, installing XNA Game Studio 4.0, 6
Draw method, 16-19
  game loop, 36–38
  virtual methods, Game class, 32
DrawIndexedPrimitives, 80, 82
drawing
  2D objects to screen, 14–16
    animation, 20–21
    controlling state, 21–25
    Draw method, 16–19
    moving things around, 19–20
  avatars with AvatarRenderer, 246–248
  with custom effects, 177–178
  primitives, 71–72
    DrawIndexedPrimitives, 80, 82
    DrawPrimitives, 78–80
    DrawUserIndexedPrimitives, 75–78
    DrawUserPrimitives, 72–75
    primitive types, 71
    vertex types, 71
DrawModel call, 101-102
DrawModeViaMeshes, 101
DrawOrder property, 40
DrawPrimitives, 78-80
DrawString method, 25
DrawUserIndexedPrimitives, 75-78
DrawUserPrimitives, 72-75
DrumKit, 332
DualTextureEffect, 105, 122-124, 160, 308
dynamic sound effects, generating, 371-374
DynamicSoundEffectInstance, 370-371

E

effect files, 173
  global variables, 174
  sampler states, setting, 183–184
  vertex structures, 174–177
effect interfaces, 121-122
effect states, 209-210
  alpha blending, 211–213
Effect type, 177
effects. *See* custom effects
Effects collection, 96
ElapsedGameTime, 35
emissive lighting, 198
EmissiveColor, 114
EnableDefaultLighting, 111
EndMark, 299
enumerating microphones, 368
enumerations
  SpriteEffects, 19
  SpriteSortMode, 22
EnvironmentMapAmount parameter, 127
EnvironmentMapEffect, 105, 124-127, 308
EnvironmentMapSpecular parameter, 126
event-based input versus polling, 312
expressions, avatar animation, 245
extensions, debugging content pipeline, 222

F

faking shadows with depth buffer and render targets, 158-161

feedback, Windows Phones, 342, 344
  - vibration, 351

fields of matrix, 59

FindSession method, 431-432

FindSessionDraw, 432

FindSkeleton method, 279

fire and forget audio playback, 354-355

FlattenSkeleton method, 279

FlattenTransforms helper method, 135

FlightStick, 332

floats, 90

FMRadio, 459-460

fog, 202-206
  - BasicEffect, 119-121

FontName node, 25

FresnelFactor parameter, 126

FriendRequestReceivedFrom, 407

FriendRequestSentTo, 408

G

Game class, 29, 32, 413
  - methods, 33
    - ResetElapsedTime, 33
    - Run, 33
    - RunOneFrame, 33
    - SuppressDraw, 33
  - properties, 34
    - InactiveSleepTime, 34
    - IsActive, 34
    - IsFixedTimeStep, 34
    - LaunchParameters, 34
    - TargetElapsedTime, 34
  - time, 34-36
  - virtual methods, 32
    - Draw, 32
    - Initialize, 32
    - OnActivated, 33
    - OnDeactivated, 33
    - ShowMissingRequirementMessage, 33
    - Update, 32

game loop
  - Draw, 36-38
  - Update, 36-38

Game Studio 2.0, 3

GameComponents, 38-40

GameDefaults, 405-406

GameLobbyUpdate method, 439

gamepad (Xbox 360), 324-325
  - connections, 333
  - controllers, 332-333
  - moving sprites, 329-331
  - reading gamepad state, 325-326
    - gamepad buttons, 326-328
    - gamepad direction pad, 328
    - gamepad thumb sticks, 329
    - gamepad triggers, 329
  - thumb stick dead zones, 332

gamepad state, reading, 325-326
  - gamepad buttons, 326-328
  - gamepad direction pad, 328
  - gamepad thumb sticks, 329
  - gamepad triggers, 329

GamePad.GetState, 327

GamePadButtons, 326-328

GamePadDPad, 328

GamePadThumbSticks, 329

GamePadTriggers, 329

GamePadType, 332

Gamer Services, 391
  GameDefaults, 405-406
  gamers, 402-405
  GamerServicesComponent, 391-392
  Guide class, 392
    platform-specific guide functionality, 397-402
    trial mode, 392-396
  IsFriend, 407-408
  Presence property, 406
  Privileges property, 406-407
  profiles, 402-405

GamerObject, 427

GamerPresence object, 406

GamerProfile, 404

gamers, 402-405
  GameDefaults, 405-406
  IsFriend, 407-408
  Presence property, 406
  Privileges property, 406-407

GamerServicesComponent, 391-392

Gamertag, 426

games
  multiplayer. *See* multiplayer games
  writing first game, 11-12

GameServices, 409

GameState, 413

GameTime object, 35

garbage collectors, 289, 291-292

general performance, 287-289
  garbage collectors, 289, 291-292
  multithreading, 292-293

generating
  sound effects, 371-374
  vectors, 301

GeoCoordinate, 350

geometry, 43

GestureSample, 340-341
  displaying data, 341-342

GestureType, 339-340

GetFriends, 407

GetGamerPicture, 403

GetState method, 321

GetUserStoreForApplication method, 378

global variables, effect files, 174

GPU (graphics processing unit), 62

GPU bound, 294, 307

graphics. *See* 2D graphics; 3D graphics

graphics cards, 62

graphics performance, 293, 295

graphics pipeline, 61
  backface culling, 63
  blending, 65
  color value, 65
  graphics cards, 62
  pixel shaders, 64
  pixel tests, 64
    depth test, 65
    scissor test, 64
    stencil test, 64
  rasterization, 64
  vertex shaders, 62
    projection space, 63
    view space, 62
    world space, 62
  viewport clipping, 63

graphics processing unit (GPU), 62

graphics profiles
  HiDef profile, 66-67
  platform capabilities, 66
  Reach profile, 66

graphics resources, tombstoning, 484-485

GraphicsAdapter class, 67-68

GraphicsDevice, 17, 30, 69-70
  creating, 70
  reference devices, 71

GraphicsDeviceManager, 14, 30

GraphicsProfile, 30, 66, 455
  HiDef, 455, 457
  Reach, 455, 457

Guide class, 392
  platform-specific guide functionality, 397
    messaging and signing in, 400–402
    notifications, 397
    players, 398–400
  trial mode, 392–396
  Windows Phone 7, 397

Guide.DelayNotifications method, 397

Guitar, 333

H

HiDef profile, 3
  graphics profiles, 66–67
  GraphicsProfile, 455, 457

history of XNA Game Studio, 1-3

HLSL (High Level Shading Language), 172

homogeneous divide, 63

I

IAvatarAnimation, 246

identity matrix, 3D graphics, 54-55

images. *See* 2D graphics; 3D graphics

InactiveSleepTime, 34

IndexBuffer, 82

Initialize virtual methods (Game class), 32

input, 311
  analog, 312
  chatpads, 313
  digital, 312
  event-based input versus polling, 312
  keyboards, 312–313
    moving sprites, 315–316
    onscreen keyboards, 316, 318–320
    reading keyboard state, 313–315
  mouse, 320
    moving sprites, 322–324
    reading mouse state, 320–322
    setting position, 324
  multitouch. *See* multitouch input
  polling versus event-based input, 312
  Xbox 360 gamepad, reading gamepad state, 325–326
    gamepad buttons, 326–328
    gamepad direction pad, 328
    gamepad thumb sticks, 329
    gamepad triggers, 329

input vertex structures, 174

installing XNA Game Studio 4.0, 5
  App Hub membership, 6–7
  downloading tools, 6
  XNA Game Studio Connect, 9–10
  XNA Windows Phone Developer Registration tool, 11

integers, 90

interacting with objects (avatars), 260-263

interfaces, avatar animation, 246

InverseDestinationColor, 146

InverseSourceColor, 146

InviteAccepted, 408

InviteREjectedProperty, 408

IsActive, 34

IsFixedTimeStep, 34
IsFriend, 407-408
IsFullScreen, 30
isolated storage, 375-377
  IsolatedStorageFile object, 379–380
  saving and loading data, 377–379
IsolatedStorageFile object, 379-380
IsTrial, 394
IsVisualizationEnabled, 443

J–K

jaggies, 31
joining available network sessions, 435-436
JoinSession method, 435-436

key lights, 111
keyboard state, reading, 313-315
keyboards, input, 312-313
  moving sprites, 315–316
  onscreen keyboards, 316, 318–320
  reading keyboard state, 313–315

L

launchers, Windows Phone 7, 463-472
LaunchParameters, 34
Level object, 227-228
libraries, time ruler, 306
lightDirection, 193
lighting, 186
  ambient lighting, 108, 186–190
  BasicEffect, 108–114
  calculating, 110
  diffuse lighting, 192–196
    multiple lights, 196–197
    oversaturation, 197
  directional lights, 109–111
  emissive lighting, 198
  fog, 202–206
  key lights, 111
  modifying avatars, 248–249
  point lights, 206–209
  specular color, 113
  specular highlights, 112
  specular lighting, 199
    Blinn-Phong shading, 200–202
    Phong shading, 199
  triangle normals, 190–192
LinearClamp, 168
LinearWrap, 168
LoadContent method, 15, 38-39
loading
  avatars while drawing, 247–248
  data, isolated storage, 377–379
  loose files from projects, 388–390
location service, Windows Phones, 348
  reading location data, 348–351
looping audio playback (SoundEffect), 356

M

main menu, multiplayer games, 412-416
MainMenuDraw, 416
Managed DirectX, 2
managing performance, 295-303, 305
  performance measurement tools, 306–307
manipulating vectors with matrices, 59, 61
matrix
  3D graphics, 53–54
    combining matrix transforms, 58
    identity, 54–55
    manipulating vectors, 59, 61
    rotation, 56–57
    scale, 57
    translation, 55

- fields of, 59
- methods, 60-61
- properties of, 60

MeasureString method, 27

media, 441
- MediaPlayer, 442-443
- songs
  - metadata and, 443-444
  - playing, 441-442

media enumeration, 444-448

media libraries, 444-448

MediaPlayer, 442-443, 449

Meshes, 96-98

messaging, Guide class, 400-402

metadata, songs and, 443-444

methods, 86
- Begin, 22
- Draw, 16-19
- DrawString, 25
- Game class, 33
  - ResetElapsedTime, 33
  - Run, 33
  - RunOneFrame, 33
  - SuppressDraw, 33
- LoadContent, 15
- MeasureString, 27
- ToRadians, 19

microphones
- enumerating, 368
- reading data, 369
- recording with, 368-371

Microsoft Cross-Platform Audio Creation Tool (XACT), 353

Microsoft.Xna.Framework.Net, 410

mipmapping, 169

ModelBone object, 98

models, 95
- bones, 98-99
- Meshes property, 96-98
- rendering, 99-103

modifying avatar lighting, 248-249

mouse, 320
- moving sprites, 322-324
- reading mouse state, 320-322
- setting position, 324

mouse state, reading, 320-322

mouse window handle, setting, 324

MouseState structure, 322

moving sprites
- with accelerometer data, 346-347
- with keyboard input, 315-316
- with mouse, 322-324
- multitouch input, 337-339
- Xbox 360 gamepad, 329-331

multiplalyer networking, 409

multiplayer games, 409-410
- joining available network sessions, 435-436
- main menu and state management, 412-416
- network sessions, creating, 416-417, 424
- networking development, 410, 412
- playing, 427-430
- searching for available network sessions, 430-434
- sending player invites, 438-439
- simulating real world network conditions, 439-440

multiple lights, diffuse lighting, 196-197

multisampling, 31

multithreading, 292-293

multitouch input for Windows Phones, 334
  displaying GestureSample data, 341-342
  moving sprites, 337-339
  number of touch points, 336
  reading gestures from TouchPanel, 339-341
  reading TouchPanel device state, 334-336
  TouchPanel width, height, orientation, 337
MyContentProcessor class, 236

N

namespaces, table of, 3
.NET runtime, 289
network sessions
  joining, 435-436
  multiplayer games, 416, 418, 424
  searching for available, 430-434
networking, multiplayer, 409
networking development, multiplayer games, 410, 412
networks, simulating real world network conditions, 439-440
NetworkSession, 424-425
NetworkSession.Create, 432
NetworkSession.Find, 432
NetworkSession.SimulatedLatency property, 439
NetworkSession.SimulatedPacketLoss property, 440
new projects, 29-32
nonuniform scale, 57
normals, triangle normals, 190-192
notifications, Guide class, 397
NumVertices, 97

O

objects
  Content, 15
  GraphicsDevice, 17
  interacting with (avatars), 260-263
  SpriteBatch, 16
    animation, 20-21
    controlling state, 21-22, 24-25
    drawing, 16-19
    moving things around, 19-20
OnActivated virtual methods (Game class), 33
OnDeactivated virtual methods (Game class), 33
onscreen keyboard, 316, 318-320
OpaqueDataDictionary, 237
opening XACT, 360
orthographic (projection matrix), 93
otherPlayer, 398
output vertex structures, 175
oversaturation, diffuse lighting, 197

P

packetReader, 429
Pan, 356
ParentBone, 98, 251
party invites, 439
party user interfaces, 400
passes, effect files, 176
performance
  general performance, 287-289
    garbage collectors, 289, 291-292
    multithreading, 292-293
  graphics performance, 293, 295
  managing, 295-303, 305
    performance measurement tools, 306-307

perspective, projection matrix, 89-92

phone-specific features, handling tombstoning, 482-484

Phong shading, specular lighting, 199

photo choosers, Windows Phone 7, 472

photos, rendering, 473

Pitch, 356

pixel shaders, 64, 176, 182

pixel tests, 64

- depth test, 65
- scissor test, 64
- stencil test, 64

platform capabilities, graphics profiles, 66

PlatformContents, 156

playback, DynamicSoundEffectInstance, 370-371

player invites, sending, 438-439

players, Guide class, 398-400

playing

- multiplayer games, 427-430
- multiple animations, 249-252
- songs, 441-442
- sound effects, 353
  - SoundEffect. *See* SoundEffect
  - XACT. *See* XACT

PlayingGameUpdate, 427-428

point versus direction (vectors), 46

point lights, 206-209

polling versus event-based input, 312

PreparingDeviceSettings, 31

Presence property, 406

PresenceMode, 406

PresentInterval, 69

PreserveContents, 156

primitives, drawing, 71-72

- DrawIndexedPrimitives, 80, 82
- DrawPrimitives, 78-80
- DrawUserIndexedPrimitives, 75-78
- DrawUserPrimitives, 72-75
- primitive types, 71

private void CreateSession(GameType gameType), 425

private void CreateSessionDraw(), 418

Privileges property, 406-407

ProcessAnimation, 282

processAnimations helper methods, 136

profilers, 307

profiles, 402-405

- GameDefaults, 405-406
- IsFriend, 407-408
- Presence property, 406
- Privileges property, 406-407

projection matrix, 88

- orthographic, 93
- perspective, 89-92

projection space, 63

projects

- loading loose files from, 388-390
- new projects, 29-32
- recreating on Xbox, 380-382
- XACT, creating new, 360

properties

- content processors, 221
- Game class, 34
  - InactiveSleepTime, 34
  - IsActive, 34
  - IsFixedTimeStep, 34
  - LaunchParameters, 34
  - TargetElapsedTime, 34
- matrix, 60

Q–R

radians, converting to angles, 19

rasterization, 64

RasterizerState, 164-166

Reach profile, 3
  graphics profiles, 66
  GraphicsProfile, 455, 457

reacting to tombstoning, 480-482

reading
  acceleration data, 344
  gamepad state, 325-326
    gamepad buttons, 326-328
    gamepad direction pad, 328
    gamepad thumb sticks, 329
    gamepad triggers, 329
  gestures from TouchPanel, multitouch input, 339-341
  keyboard state, 313-315
  location data, 348-351
  microphone data, 369
  mouse state, 320-322
  TouchPanel device state, 334-336

ReceiveData, 429

recording with microphones, 368-371

reference devices, GraphicsDevice, 71

reference types, 288

ReferenceStencil value, 163

render targets
  2D avatars, 263-265
  faking shadows, 158-161

rendering
  3D. *See* 3D rendering
  models, 99-103
  photos, 473
  targets, 155-158
  text, 25-27
  video, 448-450
  visualization data, 451-453

RenderScene helper, 160

RenderState object, 141

RenderTarget2D, 155, 158

RenderTargetUsage options, 156

repeating textures, 184-186

ResetElapsedTime, 33, 36

ReverseSubtract value, 145

rotation matrix, 3D graphics, 56-57

Run, 33

RunOneFrame, 33

S

Sampler, 182

sampler states, 166-169
  setting in effect files, 183-184
  texture types, 169

SampleRate, 372

SamplerState object, 141

saving data, isolated storage, 377-379

scale matrix, 3D graphics, 57

scissor test, 64

screens, drawing 2D objects to, 14-16
  animation, 20-21
  controlling state, 21-22, 24-25
  Draw method, 16-19
  moving things around, 19-20

searching for available network sessions, 430-434

SearchQuery, 471

SecularColorPower, 201

SendDataOptions, 429

sending player invites, 438-439

sensors, Windows Phones, 342, 344

sessions, creating, 423

shader models, 172

shaders

- built-in, cost of, 307-308
- pixel shaders, 176, 182
- vertex, 182
- vertex shaders, 175

shading

- Blinn-Phong, 200-202
- Phong, 199

shadows, faking with depth buffer and render targets, 158-161

ShowGameInvite, 400

ShowMarketplace, 394

ShowMissingRequirementMessage virtual methods (Game class), 33

SignedIn event, 403

SignedInGamers, 241, 403

signing in Guide class, 400-402

SimulateTrialMode, 393

simulating real world network conditions, 439-440

Size node, 25

SkinnedEffect, 105, 127-140, 308

songs

- metadata and, 443-444
- playing, 441-442

sound effects

- generating, 371-374
- playing, 353
  - SoundEffect. *See* SoundEffect
  - XACT. *See* XACT
- recording audio with microphones, 368-371

SoundEffect, 354

- 3D audio positioning, 356-357
- adding SoundEffectInstance to games, 357-360
- adjusting Pitch, Pan, and Volume, 356
- fire and forget, 354-355
- loading from files, 354
- looping audio playback, 356
- SoundEffectInstance, 355

SoundEffectInstance, 355

- adding to games, 357-360

Spacing node, 25

specular color, 112-113

specular highlights, 112

specular lighting, 199

- Blinn-Phong shading, 200-202
- Phong shading, 199

SpecularColor, 113

SpecularLightColor, 201

SpriteBatch object, 16

- animation, 20-21
- controlling state, 21-25
- drawing, 16-19
- moving things around, 19-20

SpriteEffects enumeration, 19

sprites, moving

- with accelerometer data, 346-347
- with keyboard input, 315-316
- with mouse, 322-324
- with multitouch input, 337-339
- with Xbox 360 gamepad, 329-331

SpriteSortMode enumeration, 22

StartIndex, 97

state management, multiplayer games, 412-416

StateBlock object, 141

states
  controlling (2D graphics), 21–25
  device state
    BlendState object. *See* BlendState object
    DepthStencilState, 149–155
  device states, 141–142
    RasterizerState, 164–166
    stencil buffer, 161, 163–164
  sampler states, 166–169
    texture types, 169
static cameras, 94-95
stencil buffer, 161, 163-164
stencil test, 64
storage, 375
  API, 387–388
  containers, 382–383
  devices, 382–383
    choosing, 385
    getting, 383–386
  isolated storage, 375–377
    IsolatedStorageFile object, 379–380
    saving and loading data, 377–379
  loading loose files from your project, 388–390
  recreating projects on Xbox, 380–382
StorageContainer object, 388
streaming XACT, 366-368
StreamReader class, 226
Style node, 26
SupportedOrientations, 31
SuppressDraw, 33
SurfaceFormatColor, 31
SynchronizeWithVerticalRetrace, 31
synchronizing vertical retrace, 31
SystemLink, 410

T

Tag property (models), 96
TargetElapsedTime, 34-35
targets, rendering, 155-158
techniques, effect files, 176
tex2D function, 182
text, rendering, 25-27
Texture parameter, 126
Texture2D, 14, 169
TextureContent, 229
TextureCoordinate values, 185
TextureCube, 169
textures
  AlphaTestEffect, 124
  BasicEffect, 114–115, 118–119
  DualTextureEffect, 122–124
  repeating, 184–186
  sampler states, 166
texturing custom effects, 180-183
  repeating textures, 184–186
  setting sampler states, 183–184
thumb sticks, dead zones (Xbox 360 gamepad), 332
time, game class, 34-36
time ruler library, 306
TimeRuler, 296
TimeSpan, 351
ToggleFullscreen, 31
tombstoning, 476, 479
  graphics resources, 484–485
  handling with phone-specific features, 482–484
  reacting to, 480–482
tools
  downloading for installing XNA Game Studio 4.0, 6

- Windows Phone Developer Registration tool, 11

ToRadians method, 19

TotalGameTime, 35

touch points, multitouch input, 336

TouchCollection, 335

TouchLocation, 335, 337

TouchPanel
- device state, reading, 334-336
- reading gestures from, 339-341
- width, height, and orientation, 337

TouchPanelCapabilities, 336

tracing content through build system, 215-216

transforms, avatar animation, 245

translation matrix, 3D graphics, 55

trial mode, Guide class, 392-396

triangle normals, 190-192

U

Update, 35
- game loop, 36-38
- virtual methods, Game class, 32

UpdateBoneTransforms, 273

UpdateOrder property, 40

updating
- avatar animation, 244
- games to use custom animation, 284-285

user input. *See* input

V

ValidateMesh call, 135

value types, 289

variables, global, 174

vector addition, 47-48

vector cross product, 50

vector dot product, 49

vector negation, 49

vector scalar multiplication, 48

vector subtraction, 48

Vector2 class, 18

Vector3, 46, 51-53

Vector4, 46

vectors, 46
- 3D graphics, 46
  - addition, 47-48
  - cross product, 50
  - dot product, 49
  - negation, 49
  - point versus direction and magnitude, 46
  - scalar multiplication, 48
  - subtraction, 48
  - Vector4, 46
  - XNA Game Studio, 51, 53
- generating, 301
- manipulating with matrices, 59, 61

vertex buffer, 116-117

vertex color, 179-180
- BasicEffect, 115-118

vertex shaders, 62, 175, 182
- projection space, 63
- view space, 62
- world space, 62

vertex structures, effect files, 174-177

vertex types, drawing primitive types, 71

VertexBuffer, 78-79

VertexDeclaration, 116

VertexElement, 116

VertexOffset, 97

vertical retrace, synchronizing, 31

VibrateController, 351

vibration, Windows Phones feedback, 351

video, rendering, 448-450

VideoPlayer object, 449

view matrix, 87-88

view space, 62

viewport clipping, 63

virtual methods, Game class, 32
  Draw, 32
  Initialize, 32
  OnActivated, 33
  OnDeactivated, 33
  ShowMissingRequirementMessage, 33
  Update, 32

visualization data, rendering, 451-453

visualizations, 451-453

visualizers, 442

Volume, 356

W

wave files, adding to XACT projects, 361, 363-364

Wheel, 332

Windows desktop runtime, 291

Windows Game SDK, 1

Windows Phone 7
  choosers, 472–476
  FMRadio, 459–460
  Guide class, 397
  launchers, 463–472
  writing first game, 12

Windows Phone Developer Registration tool, installing XNA Game Studio 4.0, 11

Windows Phones
  acceleration data using accelerometer, 344–346
  feedback, 342, 344
    vibration, 351
  location service, 348
    reading location data, 348–351
  multitouch input, 334
    displaying GestureSample data, 341–342
    moving sprites, 337–339
    number of touch points, 336
    reading gestures from TouchPanel, 339–341
    reading TouchPanel device state, 334–336
    TouchPanel width, height, and orientation, 337
  sensors, 342, 344

WireFrame, 165

world space, vertex shaders, 62

Wrap, 169

wrap texture, 168

writing first game, 11-12

X–Z

X axis, 13

XACT (Microsoft Cross-Platform Audio Creation Tool), 353, 360
  adding wave files to projects, 361, 363–364
  creating new projects, 360
  opening, 360
  sound playback using cue, 364–366
  streaming, 366–368

Xbox, recreating projects, 380-382

Xbox 360
- chatpad input, 313
- devices, 382
- game data, 382
- shader models, 172
- writing first game, 11

Xbox 360 gamepad, 324-325
- connections, 333
- controllers, 332-333
- moving sprites, 329-331
- reading gamepad state, 325-326
  - gamepad buttons, 326-328
  - gamepad direction pad, 328
  - gamepad thumb sticks, 329
  - gamepad triggers, 329
- thumb stick dead zones, 332

XNA Game Studio
- 3D graphics, 41-42
- vectors, 51, 53

XNA Game Studio 4.0, installing, 5
- App Hub membership, 6-7
- downloading tools, 6
- Windows Phone Developer Registration tool, 11
- XNA Game Studio Connect, 9-10

XNA Game Studio Connect, installing XNA Game Studio 4.0, 9-10

XNA Game Studio Device Center, 10

Y axis, 13

This page intentionally left blank

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari Books Online

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press QUE PRENTICE HALL SAMS | Safari Books Online

Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos

Joomla! A User's Guide
BARRIE M. NORTH
SCOTT MUELLER'S UPGRADING AND REPAIRING PCs 18TH EDITION
CCENT/CCNA ICND1 Official Exam Certification Guide
THE RAILS WAY
OBIE FERNANDEZ
ASP.NET 3.5 UNLEASHED

Safari Books Online

FREE TRIAL—GET STARTED TODAY!
www.informit.com/safaritrial

Find trusted answers, fast

Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.

Master the latest tools and techniques

In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

WAIT, THERE'S MORE!

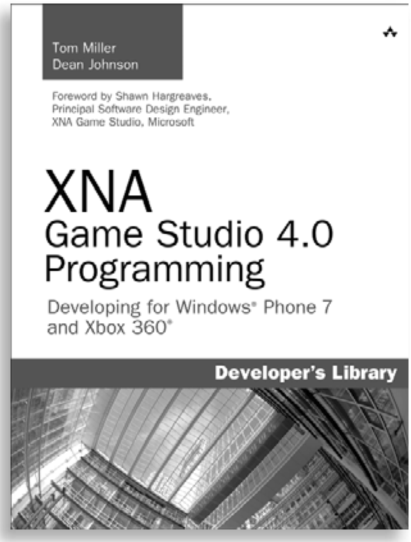
Keep your competitive edge

With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.

Stay current with emerging technologies

Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.

Addison Wesley
AdobePress
ALPHA
Cisco Press
FT Press FINANCIAL TIMES
IBM Press
lynda.com
Microsoft Press
New Riders
O'REILLY
Peachpit Press
PRENTICE HALL
que
Redbooks
SAMS
sas Publishing
Sun microsystems
Wharton School Publishing
WILEY


Tom Miller
Dean Johnson
Foreword by Shawn Hargreaves, Principal Software Design Engineer, XNA Game Studio, Microsoft
XNA
Game Studio 4.0
Programming
Developing for Windows® Phone 7 and Xbox 360®
Developer's Library

FREE Online Edition

Your purchase of ***XNA Game Studio 4.0 Programming*** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, Que, and Sams.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

Activate your FREE Online Edition at www.informit.com/safarifree

STEP 1: Enter the coupon code: GCIVNCB.

STEP 2: New Safari users, complete the brief registration form.
Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Safari Books Online

Addison Wesley · Adobe Press · ALPHA · Cisco Press · FT Press · IBM Press · lynda.com · Microsoft Press · New Riders · O'REILLY · Peachpit Press · PRENTICE HALL · que · Redbooks · SAMS · sas Publishing · Sun microsystems · Wharton School Publishing · WILEY

www.it-ebooks.info